Weak-to-Strong Memory Transfer Is Negative: Predecessor Anchoring in Frontier GPU-Kernel Agents

Lindell S. Cumes

QWNTL Labs

Project page: https://qwntl.com/paper-predecessor-anchoring

Abstract

When an agent harness upgrades its worker model, the new model starts from a memory built from its predecessor's work. ACE grows such memory as a playbook from an agent's own trajectories and lists sharing it across models among its advantages. Evolving to Match Capabilities (arXiv 2610.33024) reports that human-crafted skills lift a frontier model by 31.1 points and cost the weakest model 8.8. On KernelBench, weak-to-strong memory transfer is negative. In the first randomized, replicated test of memory carried across a model upgrade, a store built from Claude Opus 4.8's verified kernel work lowered Claude Opus 5.5's log speedup by 0.103 (95% CI 0.048–0.163; 92 problems, 1,288 episodes). It raised Opus 4.8's by 0.256. A confirmation on 28 problems outside calibration reproduced both signs. Six ways of serving the store, compared head to head, also left Opus 5.5 below its unaided level, and so did its own decision to read the store. Shown the store, Opus 5.5 gives up its own CUDA C++ kernels for its predecessor's library-plus-Triton designs. Predecessor anchoring is a stronger reader's move onto a weaker writer's approach, with the loss that follows. The same store holds GPT-6 Astra below an information-free control. The store's effect splits exactly into a within-approach term and an adoption term, the shift toward the stored approach times the gap between that approach and the reader's own. That gap depends on the reader and the problem it faces, and a certificate on an item's source problem cannot fix the sign of the item's effect for a new reader. Six evaluator-verified technique cards, each stating a technique and when it applies, carry an approach above the reader's own. On 26 KernelBench problems they lifted ACE's own playbook from $1.95\times$ to $3.29\times$ mean capped speedup. With them the store reached $3.67\times$ on 10% of ACE's input tokens.

Release. https://qwntl.com/paper-predecessor-anchoring hosts what the readers were shown (the hour-24 and hour-36 stores as served, the six technique cards and the task prompt), the protected evaluator with its attack suite, and per-episode results for all seven designs. The code is released as run, with internal identifiers renamed and host paths removed.

1 Introduction

The long-running harness studied here keeps a store of what its worker has verified on KernelBench [21]: the kernels that passed a protected evaluator, and the recipes and pitfalls written from the worker's successes and failures. Claude Opus 5.5 replaced Claude Opus 4.8 as the worker at hour 36. The harness kept the whole store, written at the old model's level, and every item in it was still correct.

Dynamic Cheatsheet [27] and ReasoningBank [22] grow stores from the reader's own trajectories. ACE [34] grows a playbook the same way. It reports a 10.6% average gain over baselines on AppWorld. Randomized Block Curation values lessons by experiments on practice tasks, with the same model writing and reading them, and finds self-written

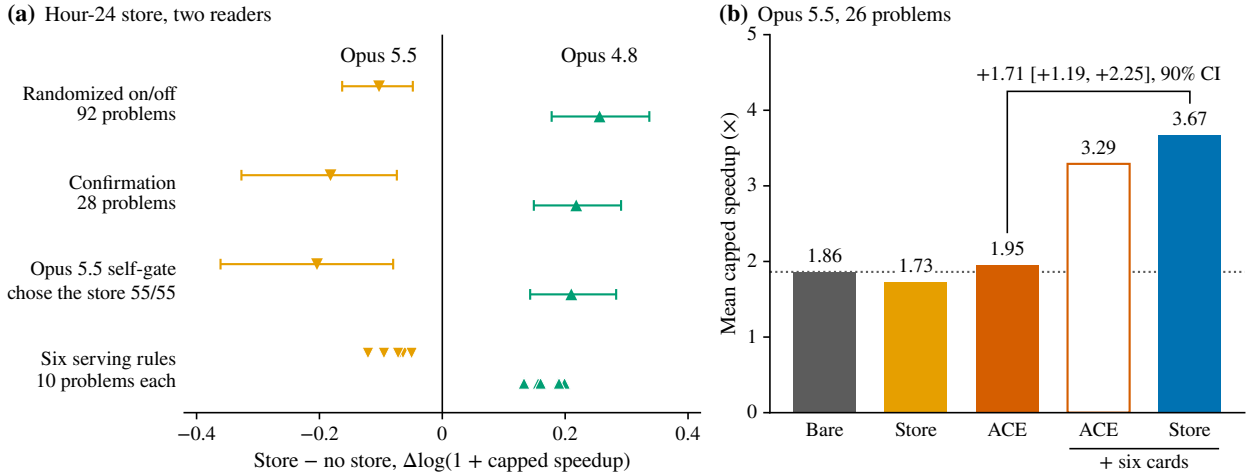

Figure 1: **Weak-to-strong memory transfer is negative under every serving rule, and technique above the reader turns it positive.** (a) Store minus no store in $\log(1 + \text{capped speedup})$ on KernelBench for the hour-24 store built from 18 hours of Claude Opus 4.8 kernel work, read by Claude Opus 5.5 and by Opus 4.8, whose work built it. Horizontal bars are 95% problem-bootstrap intervals. Every row shares the x axis. Randomized: the whole store on or off at random, 1,288 episodes over 92 problems (77 for Opus 4.8). Confirmation: 28 problems outside calibration, 1,120 episodes. Self-gate: the store served wherever a zero-shot Opus 5.5 prompt chose it, which was 55 of 55 problems. Six serving rules: point estimates for the whole store and for the five rules of Table 7 that select or annotate items, two of them Opus 5.5's own choices, on 10 held-back problems with 4 repetitions. (b) Claude Opus 5.5's mean capped speedup on 26 KernelBench problems outside the card-search trials, 720 episodes. The dotted line is bare. The bracket gives the paired difference of the store with six cards from ACE's playbook, with its 90% task-bootstrap interval. The outlined bar adds the same six cards to ACE's playbook, an arm added by preregistered amendment.

playbooks up to 69.6 points below no playbook [24]. Evolving to Match Capabilities finds that human-crafted skills lift a frontier model by 31.1 points and lower the weakest model by 8.8 [33]. In that study, guidance pitched at a strong reader failed a weaker one. Memory Tax prices memory as its benefit minus a tax relative to naive retrieval [9], and MetaKernelBench measures skill transfer between kernel languages on the same problem, from -19% to $+29\%$ [5].

Most of these studies serve memory to the model that wrote it or carry guidance down the capability ladder. The closest, Memory Tax, builds memory with each of three backbones and reads each memory with all three. It reports the reader swap as a change in accuracy against naive retrieval. Weak-to-strong memory transfer, the case these studies leave open, is the effect of memory built from a weaker model's verified work on a stronger reader. We measure it by exposing the stronger reader to such memory at random and comparing it with the same reader without memory.

We froze the harness's store at hour 24, after 18 hours of the predecessor's kernel work, and switched it on or off at random for each reader (Figure 2); a protected evaluator scored every kernel the reader submitted at strict fp32 tolerance by its *capped speedup* over PyTorch eager. The cap is $10\times$, and an incorrect kernel scores zero. *Log speedup* is $\log(1 + \text{capped speedup})$.

We report the first randomized, replicated measurements of memory carried across a model upgrade, and the first to place the upgraded model below its level without memory (Section 4.2). Over 1,288 randomized episodes, the store lowered Opus 5.5's log speedup by 0.103 $[0.048, 0.163]$ across 92 problems. It raised Opus 4.8's by 0.256 $[0.178, 0.337]$ across 77 (Figure 1a). A confirmation test on 28 problems outside calibration reproduced both signs. The store anchored the stronger reader, which moved onto its predecessor's approach and fell below its own unaided level.

A preregistered head-to-head then served the store through rules that choose items, from the harness's provenance filter and a ridge-regression selector to Opus 5.5's own choices from per-item randomized outcomes, and through per-item evidence tags that annotate them. Every one of them left Opus 5.5 below bare and lifted Opus 4.8. Asked whether the store would help, Opus 5.5 said yes on 55 of 55 problems. On 21 further problems, a second pass spent on the store fell below an information-free control for both Opus 5.5 and GPT-6 Astra (Section 4.7).

By the law of total expectation, the store's effect splits exactly into two terms (Section 3.4), and neither involves item

correctness. The adoption term is the shift toward the stored approach times the approach gap, which is the value of that approach in the reader's hands minus the value of the reader's own, and the within-approach term is the change in how well the reader executes a given approach. When the within-approach term is zero and adoption increases, the effect has the sign of the gap (Proposition 3.4). The evaluator certifies each item on the problem where it was found, and the gap depends on the reader and on the problem it now faces. On KernelBench L2_074, Opus 5.5 alone writes one CUDA kernel at $8.86\times$. Shown the store, it writes its predecessor's cuDNN-plus-Triton design, whose library call caps it at $1.34\times$. The decomposition predicts why item selection and verification fail, and we confirmed it on the backend channel and on six serving rules (Sections 4.4 to 4.6).

The remedies studied here raise the gap or change adoption, and six evaluator-verified technique cards raise the gap by carrying an approach above the reader (Section 5). With them the store reached $3.67\times$ mean capped speedup on 26 KernelBench problems, against $1.95\times$ for ACE's playbook grown on the same stream (Figure 1b). The seed card is an fp32-accurate GEMM on bf16 tensor cores. Late delivery changes adoption by serving the unchanged store after the reader's first checked attempt. It lifted Opus 5.5 by 0.106 log speedup above no memory.

2 Related work

Agent memory holds the reader fixed. Reflexion [26] and ExpeL [36] keep text distilled from past trials, verbal feedback between trials in one and insights from trajectories in the other. Voyager [30] and Agent Workflow Memory [32] keep reusable procedures, as executable skills verified in the environment and as induced workflows respectively. MemGPT and Mem0 persist facts and preferences [6, 23]. Dynamic Cheatsheet [27] and ReasoningBank [22] build a store from the reader's own experience. So does ACE [34], which grows an evolving playbook through generator, reflector, and curator roles. It reports a 10.6% average gain over baselines on AppWorld. Its main experiments use the same model in all three roles. That design rules out transfer from a stronger reflector to a weaker generator, and its weaker-reflector test on FiNER (+5.9 points) keeps the generator as the reader. ACE lists sharing contexts across models among the advantages of evolving contexts. No ACE experiment serves a playbook to a model other than the one that generated it. We served the store studied here to a model it was not built from, and when we grew ACE's playbook on the same stream with its roles reassigned, it reached $1.95\times$, against $3.67\times$ for this store with verified technique (Section 5).

Valuing memory per reader. Randomized Block Curation values lessons by experiments on practice tasks and finds self-written playbooks up to 69.6 points below no playbook [24]. There the growth run decides the sign, and the model that writes the lessons also reads them. Its agents' own choice of which lessons to keep scored 53.8 against 54.1 for no playbook. The successor's 55-of-55 self-gate here is the same failure. Memory Tax splits the net value of memory into a benefit and a tax relative to naive retrieval and finds that many dedicated memory systems fall below naive RAG. It also builds memory with each of three backbones and answers over it with all three, and changing the reader moves memory systems more than it moves naive retrieval [9]. Its reference is retrieval over raw history in conversational question answering, and it measures the reader swap as an accuracy drop on memory that is not verified. Raw-history retrieval leads agentic memory in six of seven conversational settings [1]. Sentry keeps a failure playbook outside the context and retrieves lessons when it detects a matching failure; placing the full playbook in context lowered WebShop reward from 0.364 to 0.296 with one model in every role [12]. Beyond Gain retires a skill for a model once that model succeeds without it, at no detectable cost on SkillsBench (-0.2 points) [31]. Many Steps into One shows that a memory goes stale when the facts beneath it change [37]. Apart from Memory Tax, these studies hold the reader fixed. After an upgrade, which replaces the reader and keeps the memory, the predecessor's store is verified and on task and still harms the new reader, a state for which none of these taxonomies has a category. We value memory by randomized exposure for each reader.

Capability-matched guidance. Evolving to Match Capabilities finds that human-crafted skills lift a frontier model by 31.1 points and lower the weakest model by 8.8 on 103 selected task–skill pairs [33]. In 86% of its failed weak-model runs the violated requirement is already stated in the skill. A capable external rewriter lifts weak executors by 5.1–14.1 points. BiPA moves learned skills from DeepSeek-V4.1-Flash to a lower-scoring reader and finds them below the seed, as that reader's own learned skills are [38]. Both carry guidance down the capability ladder or across it; we measure the upward case, with the guidance written from the weaker model's own verified work, and find its sign negative. MILO argues that harness design must be repeated as models change [11]. HarnessTTT finds that learned harnesses transfer to

stronger executors [17]. In that work the part that transfers upward is a harness procedure. The store studied here holds the predecessor's kernels, with recipes and pitfalls written from its work, each at its writer's level.

Kernel generation and transfer. KernelBench asks models to replace PyTorch reference modules with faster custom GPU kernels [21]. Its own in-context study is the closest precedent for the sign pattern: three hand-written exemplar kernels lowered o1's fast_1 at every level (10% to 6%, 24% to 16%, 12% to 8%) and raised Llama 3.1-70B's at Level 1 (3% to 6%). The exemplars were written by people and scored in one shot. Here the memory is the predecessor's own evaluator-verified work, and we randomized both readers on the same problems; the decomposition of Section 3.4 explains the sign. MetaKernelBench measures same-problem transfer between two kernel languages with one model as source and reader and finds lifts from -19.1% to $+29.3\%$; in that study a source attempt below the target's own harms 53.8% of comparisons, and transfer between models is left to future work [5]. METR's measurement of kernel agents found exploitable loopholes, from cacheable outputs and CUDA streams to reuse of the reference's memory [18], and Lange et al. built a hardened benchmark in response [15]. KernelBench-M audits the official check at a tolerance of 10^{-2} and finds that it misses 16.9% of witnessed faults and 78.6% of precision faults [8]. The evaluator that certified every stored item (Section 4.5) checks fp32 at the KernelBench repository's current default, 10^{-4},[1] 100 times tighter than that check and 10 times tighter than KernelBench-Verified's 10^{-3} [35, §2.5]. It must reject 21 attacks before any arm runs.

Anchoring and distraction. Jones and Steinhardt showed that erroneous code in a prompt anchors Codex toward that code [13]; every stored kernel here is correct and verified. Scaling Discovery Requires Forgetting finds search agents anchored on their own past attempts and restores progress by restarting from the best candidate, up to 15% over parallel agents [28]. In this paper the anchor is a predecessor's verified kernel, and it holds a stronger model below its own bare level. Few-shot prompting degrades DeepSeek-R1 [7]. Irrelevant context distracts models on GSM-IC [25]. Red herrings fixate them on the Only Connect Wall dataset [19]. The store studied here is on task, and it helps Opus 4.8, whose kernels built it. Weak-to-strong generalization fine-tunes a strong model on weak labels and finds it above its supervisor and below its own ceiling [4]. In our experiments the weak supervision arrives as memory in context, and the strong reader falls below its own unaided level.

The closest human analogue is the Einstellung effect, in which a familiar procedure blocks a shorter one; in chess a known solution holds experts to the level of much weaker players [2, 3, 16]. In the expertise reversal effect, guidance that helps novices harms experts [14]. Across a model upgrade the same reversal appears, and the guidance is written from the novice's own work.

3 Carrying memory across a model upgrade

A harness is the system that streams tasks to a worker model and keeps a store built from the model's checked work. We define an *agent* as a harnessed model instance. Each store has a *writer*, the model whose work it was built from, and a *reader*, the model it is served to. Before an upgrade the two coincide; after it they differ, and the reader is the stronger model.

3.1 Problem statement and information flow

Task, agent, budget, and objective. A KernelBench problem is a tuple $\tau_p = (d_p, x_p^0, V)$: the description d_p is a PyTorch reference module `Model`, the start state x_p^0 is an empty `ModelNew`, and the verifier V is the protected evaluator of Section 3.2, which returns the capped speedup $s \in [0, 10]$, zero if the submission is incorrect. An agent is a pair $A = \langle M, H_\pi \rangle$ of a model M and a harness H_π: the kernel agent loop of Section 3.3 with a serving policy π that decides what part of a store S the model reads, and when. We fix the budget B at one episode of at most five turns and one protected submission, with at most 32,000 output tokens per reply and 2,400 seconds of wall clock. Arms are matched in episodes per problem and share every cap. The objective is work at fixed budget: the mean over problems of $\mathbb{E}[\log(1 + s)]$ for one episode, and the value of serving is its difference from the same agent with no store (Definition 3.1).

[1] https://github.com/ScalingIntelligence/KernelBench at commit 423217d9, `src/kernelbench/eval.py`, lines 94–95; versions v0 and v0.1 used 10^{-2}.

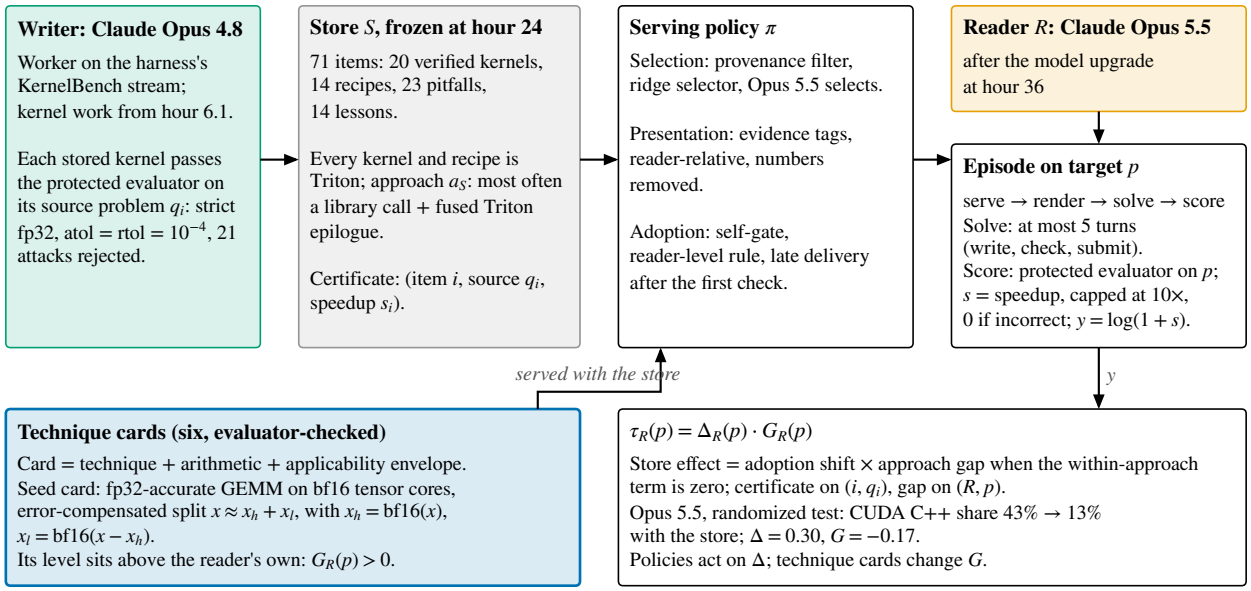

Figure 2: **The evaluator certifies each stored kernel on its source problem, and the store's effect on a new reader is set on the target.** The Claude Opus 4.8 worker's kernels pass the protected evaluator on their source problems q_i and enter the store frozen at hour 24; its recipes, family pitfalls, and four lessons were written from Opus 4.8's work by the auxiliary model, Claude Opus 5.5. A serving policy π chooses the items and how they read and decides whether and when the reader sees them (Algorithm 1); after the upgrade at hour 36 the reader solves the target p, and the evaluator scores it there. Lower right: the store effect is the adoption shift times the approach gap (Proposition 3.4), with the backend channel measured in the randomized test. Lower left: a technique card carries an approach above the reader and has a positive gap (Section 5).

Information-flow contract. For every evaluation episode we froze the model weights and the evaluator, together with what the arms draw on, the store, the technique cards and the problem splits; every episode starts from the same snapshot, and no item is written while an arm is scored. The serving decision is the only thing that changes between arms: which items the reader sees and how they are rendered, and whether and when the store arrives, in the opening prompt or after the reader's first local check. Inside an episode the reader sees the reference module, the rendered store if served, and its own local checks (compilation, the rules, and correctness on one public input, with no timing); the protected evaluator's score arrives only after submission. The store's writer saw protected scores when the store was built; policies fitted to outcomes (the ridge selector, Opus 5.5's item and reader choices, the reader-level rule) saw randomized outcomes from earlier runs, and each was frozen before its evaluation episodes began.

3.2 Harness components

Long-running harness. The harness ran for 44 hours, streaming KernelBench tasks to a worker alongside AppWorld, τ^2-bench and coding tasks. Kernel work began at hour 6.1. Claude Opus 4.8 was the worker until hour 36 and Claude Opus 5.5 after it, while an auxiliary model, Opus 5.5 throughout, wrote recipes and pitfalls from the worker's verified solutions and failed attempts.

Store items. A store item is a record that pairs a text with an identifier, a kind and a section label. One recipe in the hour-24 store reads, in full:

```
id: rec-7d67d21ded  kind: recipe
label: ### Verified recipes for conv_transpose problems (speedups measured by the
protected evaluator)
text: [1.26x on ConvTranspose3d_LeakyReLU_Multiply_LeakyReLU_Max] Keep the
ConvTranspose3d itself in cuDNN via nn.ConvTranspose3d and fuse the entire epilogue
into one Triton kernel that reads the conv output once: LeakyReLU, per-channel scale,
LeakyReLU, MaxPool. Launch one thread per pooled output element over a flat index
(BLOCK=256, 1D grid). Decode n·c/od/oh/ow with div/mod, load the per-channel
```

```
## Kernel memory (items verified by the protected evaluator on training problems; may not all apply)
### Lessons about this environment (from official checker verdicts and verified outcomes)
- Checker rule:  At least one of your own kernels must actually launch during forward; a pure torch path
(even with a compiled but unused kernel) is rejected as no_custom_kernel.
- Process:  never SUBMIT code that has not passed ACTION: CHECK in this episode.  [...]
[12 further lessons]
### Pitfalls seen on earlier conv_transpose problems (wrong results or slower than eager)
- Pitfall:  Don't rewrite conv_transpose as flipped-weight F.conv2d plus a separate elementwise kernel; the
per-call weight flip and extra full-output pass make it slower.  Fuse bias into a real kernel, or skip
pointless copies.
[4 further pitfalls]
### Verified recipes for conv_transpose problems (speedups measured by the protected evaluator)
- [1.32x on ConvTranspose3d_BatchNorm_Subtract] Keep ConvTranspose in cuDNN via torch.  Fuse everything
after it into two Triton passes over the contiguous (N*C, S) view.  [...]
- [1.26x on ConvTranspose3d_Sum_LayerNorm_AvgPool_GELU] Keep ConvTranspose in cuDNN via torch and fuse only
the epilogue in Triton.  [...]
[3 further recipes:  2.29x, 1.68x, 1.06x]
### A verified kernel for a related conv_transpose problem (L2_078 ConvTranspose3d_Max_Max_Sum, 1.00x):
adapt what applies
@triton.jit def fused_kernel(x_ptr, out_ptr, N, C, D, H, W, OD, OH, OW, total, BLOCK: tl.constexpr):  [...]
  c = self.conv_transpose(x).contiguous() [...]
[1 further kernel:  L2_089, 1.06x]
```

Figure 3: **What the successor read on L2_074.** The rendered store, 10,364 characters in the harness's own headers, holds 14 lessons and 5 transposed-convolution pitfalls along with 5 recipes and 2 verified kernels. Items are verbatim; bracketed text marks omissions. The recipe for L2_074 itself was removed before serving. All five recipes route the convolution through cuDNN, and both kernels call the library convolution before their Triton code.

```
multiplier once, and unroll the 2x2x2 window with tl.static_range.  Use tl.where for
the slopes and a -1e30 init for max.  Keep all math fp32, which stays within 1e-4.
```

The recipe names its source problem and the speedup the evaluator measured there. The harness removed it before serving L2_074, its own source problem, and the successor rebuilt it from the rest of the store (Section 4.3).

Items come in four kinds, each verified in its own way. A *kernel* is the worker's submitted code for a learning problem, stored with its measured speedup after the evaluator accepted it. The auxiliary model wrote each *recipe* as a paragraph from one such kernel and tagged it with that kernel's speedup. *Pitfalls* are rules distilled from attempts on one problem family that failed or ran slower than eager. A *lesson* applies to all problems and is a checker verdict kept verbatim, a rule written from verified outcomes, or the harness's fixed process rule.

Experience store. The main store is the harness's hour-24 snapshot, 71 items built from 18 hours of Opus 4.8 kernel work. Its 20 kernels are the Opus 4.8 worker's code. Opus 5.5 wrote 14 recipes, 23 family pitfalls, and 4 of the 14 lessons from Opus 4.8's work, 41 items in all. Nine lessons are verbatim checker verdicts, and one is the process rule. All 20 kernels and all 14 recipes use Triton [29]. For each problem the harness renders all lessons, the pitfalls and recipes of the problem's family, and up to two related kernels under fixed headers, after removing each item that names the problem. Served whole, the rendered store averages 8,885 characters. Figure 3 shows what Opus 5.5 read on L2_074, and Appendix A.4 gives further items.

We use *verified* in one sense throughout: the evaluator accepted the item's kernel at strict fp32 on the item's source problem and measured its speedup there. The certificate belongs to the item and its source problem and names no reader, since only the writer's kernel was measured. It also admits kernels at parity, and one related kernel served on L2_074 is verified at $1.00\times$.

Agent loop. Readers Claude Opus 5.5 (`claude-opus-5-5`) and Claude Opus 4.8 (`claude-opus-4-8`) use the same agent loop at default settings; each writes a `ModelNew` module with CUDA C++ or Triton kernels, checks public inputs as it chooses, and submits, with at most five turns and 32,000 output tokens per reply. We compiled and timed all kernels on NVIDIA A100-SXM4-40GB GPUs.

Protected evaluator. The evaluator runs in a sandbox the agent cannot reach and requires a custom kernel, rejecting monkeypatching and fallbacks as well as streams and graphs. A trusted process that never sees candidate code produces reference outputs with fresh seeds and weights. Both processes use atol = rtol = 10^{-4} with cuDNN TF32 off, the

Input: problem p with reference module d_p; reader R; frozen store S; serving policy π.

1 $S_p \leftarrow S \setminus \{i : i \text{ names } p\}$ % no item names a held-back problem (Section 3.2)
2 $(Z, T, \textit{when}) \leftarrow \pi(R, p, S_p)$ % show or not; which items; opening prompt or first check
3 $c \leftarrow \textsc{System} \oplus d_p \oplus [\textit{when} = \text{open}]\, \textsc{Render}(T)$ % headers by kind (Figure 3)
4 **for** $t = 1, \ldots, 5$ **do** % at most 32,000 output tokens per reply
5 $(\text{code}_t, a_t) \leftarrow R(c)$; **if** $a_t = \textsc{Submit}$ or $t = 5$ **then break**
6 $c \leftarrow c \oplus \textsc{LocalCheck}(\text{code}_t)$ % compile, rules, one public input; no timing
7 **if** *when* $=$ late and this is the first check **then** $c \leftarrow c \oplus \textsc{Render}(T)$ % late delivery (Section 4.8)
8 $s \leftarrow V(\text{code}_t, p)$; **return** $y = \log(1 + s)$ % protected; fresh seeds and weights; $s \leq 10$, 0 if incorrect

Variants, as differences in line 2. bare: $Z = 0$. show-all: $T = S_p$. provenance: drop recipes whose solution R did not write. selector: items with a positive ridge estimate for R. Opus+tables, Opus+digest: items Opus 5.5 chooses from per-item outcomes. annotated: $T = S_p$, each item tagged with its measured effect for R. frontier-gated: items verified above R's bare level on their source problem. Self-gate: Z is Opus 5.5's zero-shot answer. Reader-level rule: Z depends on R alone. late: $T = S_p$, *when* $=$ late.

Algorithm 1: **One evaluation episode under a serving policy.** Every arm in the paper is this loop with a different line 2; the writer of S, the evaluator, and every cap are fixed.

KernelBench repository's current fp32 default, which is 100 times tighter than the 10^{-2} check of KernelBench v0 and v0.1 that KernelBench-M audits [8] and 10 times tighter than the 10^{-3} fp32 tolerance of KernelBench-Verified [35, §2.5]. Timing interleaves cold-cache runs on fresh inputs and rechecks the last output in each block. Before any arm runs, two legitimate kernels must pass and 21 attacks must fail. The attacks include ones that carry state across calls (reuse of prior results, caching by shape or pointer, in-place input mutation), ones that tamper with timing (side streams and timing monkeypatches), and ones that lower precision (global TF32 flags and fp16 downcasts). Each control checks that a fast submission computes the required output under the reference's conditions.

Technique library. Six technique cards make up the library, each checked by the protected evaluator. A card states a technique, with the arithmetic that makes it work and the condition under which it applies; a recipe records how the writer solved one problem. The library is headed by the seed card, an fp32-accurate GEMM on bf16 tensor cores built on the error-compensated splits of Henry, Tang, and Heinecke and of Ootomo and Yokota [10, 20], which splits each fp32 operand into two bf16 parts, $x \approx x_h + x_l$ with $x_h = \text{bf16}(x)$ and $x_l = \text{bf16}(x - x_h)$, and computes $xw \approx x_h w_h + x_h w_l + x_l w_h$, dropping the $x_l w_l$ term of order 2^{-16}. The card records its error, at most 0.34 of the 10^{-4} tolerance for reduction lengths up to 8,192, and its rate on the A100, 65–79 TFLOP/s against 19 TFLOP/s for fp32 cuBLAS, and routes itself to problems inside that envelope. A search that tested hypotheses against the evaluator admitted five more cards, three of which carry the split to convolution, transposed convolution, and attention.

3.3 Per-episode pipeline

Each episode runs four steps (Algorithm 1). (1) *Serve.* A serving policy decides whether the reader sees the store and which items. Items naming the current problem are removed, and no item names a held-back problem. (2) *Render.* The selected items are rendered under headers by kind (Figure 3) and placed in the opening prompt or, for late delivery, after the reader's first local check with an instruction to keep its own code if faster. (3) *Solve.* The agent writes, checks, and submits a kernel within five turns. (4) *Score.* The protected evaluator scores the submission by its speedup over PyTorch eager, capped at $10\times$, or zero if incorrect. Figure 2 draws the loop with the store's writer and the upgrade.

We sort the serving policies other than bare (no store) and show-all (the whole store) into three groups by what they act on. Among the *selection* policies, which choose the items, provenance filtering hides recipes from readers that did not produce the underlying solution, and a trained selector serves the items a ridge regression predicts will help. Opus+tables and Opus+digest let Opus 5.5 choose items from per-item outcome tables or their digest, and item gating serves only items whose verified speedup exceeds the reader's own level. Of the *presentation* policies, which change how the items read, annotation tags each item with its measured effect for the reader and adds a fixed diligence instruction; a reader-relative rendering frames the store as an older model's floors, and a numberless rendering removes the speedups. Among the *adoption* policies, which decide whether and when the reader sees the store, zero-shot self-gating asks Opus 5.5 whether the store would help, reader-level rules are fitted to randomized outcomes, and late delivery serves the store after the reader's first check. We froze all selections before running, and Sections 4.6 and 4.8 test the three groups against the model of Section 3.4.

3.4 Anchoring and the value of serving

For problem p, reader R, and store indicator $Z \in \{0, 1\}$, an episode ends in one submitted kernel with capped speedup $s = \min(10, T_{\text{eager}}/T_{\text{kernel}})$ if the evaluator accepts it and $s = 0$ otherwise. Its outcome is $y = \log(1 + s)$.

Definition 3.1 (Store effect)**.** The effect of serving the store to reader R on problem p, and its average over a problem set $\mathcal{P}$, are

$$\tau_R(p) = \mathbb{E}[y \mid R, p, Z = 1] - \mathbb{E}[y \mid R, p, Z = 0], \qquad \bar{\tau}_R = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \tau_R(p). \tag{1}$$

A serving policy π maps reader and problem to Z, with value $V_R(\pi) = |\mathcal{P}|^{-1} \sum_p \mathbb{E}[y \mid R, p, Z = \pi(R, p)]$. A reader-level policy depends on R alone.

We estimate both by differences of per-problem means. Capped-speedup contrasts use s in place of y and are additive. The definition separates the value of serving memory from the correctness of the solutions it contains.

Definition 3.2 (Approach and adoption)**.** The *approach* a of an episode is the design class of its submitted kernel. Let $\rho^z_R(a \mid p) = \Pr(a \mid R, p, Z = z)$ and $\mu^z_R(a, p) = \mathbb{E}[y \mid R, p, a, Z = z]$. The store's approach a_S is the class its kernels and recipes share, and $\bar{a}$ collects every other class. The *adoption shift* is $\Delta_R(p) = \rho^1_R(a_S \mid p) - \rho^0_R(a_S \mid p)$, and the *approach gap* is $G_R(p) = \mu^0_R(a_S, p) - \mu^0_R(\bar{a}, p)$, the value of the stored approach in the reader's hands minus the value of its own, both without the store.

Definition 3.3 (Predecessor anchoring)**.** Let Q be the writer of store S and R a reader whose bare performance exceeds Q's. The store *anchors* R when $\bar{\tau}_R < 0$ and the adoption shift is positive on average.

The store effect splits exactly, by the law of total expectation, into two terms:

$$\tau_R(p) = \underbrace{\sum_a \left[\rho^1_R - \rho^0_R\right](a \mid p)\, \mu^0_R(a, p)}_{A_R(p)\text{: adoption}} + \underbrace{\sum_a \rho^1_R(a \mid p) \left[\mu^1_R - \mu^0_R\right](a, p)}_{W_R(p)\text{: within approach}}. \tag{2}$$

The adoption term values the change in which approaches the reader uses at what the reader achieves with each one unaided. The within-approach term is the change in how well the reader executes a given approach.

Proposition 3.4 (Sign of transfer)**.** *If $W_R(p) = 0$ and $\Delta_R(p) > 0$, then $\tau_R(p) = \Delta_R(p)\, G_R(p)$, and $\tau_R(p)$ has the sign of $G_R(p)$.*

Proof. With the two classes a_S and $\bar{a}$, $\rho^z_R(\bar{a} \mid p) = 1 - \rho^z_R(a_S \mid p)$, so the adoption sum in Equation (2) equals $\Delta_R(p)\, [\mu^0_R(a_S, p) - \mu^0_R(\bar{a}, p)] = \Delta_R(p)\, G_R(p)$. Setting $W_R(p) = 0$ gives the result. □

The store helps where the stored approach is better than the reader's own and harms where it is worse, in proportion to how readily the reader adopts it, and item correctness does not enter the expression.

Corollary 3.5 (Downhill and uphill)**.** *Under the conditions of Proposition 3.4, suppose the stored approach has the same value in any reader's hands, $\mu^0_R(a_S, p) = v_Q(p)$, the writer's level on p, and the reader's own approach has a value $v_R(p)$ that rises with the reader's strength. Then $G_R(p) = v_Q(p) - v_R(p)$. The store helps any reader below its writer and harms any reader above it, and for a reader above its writer the harm equals the adoption shift times the reader's lead, $|\tau_R(p)| = \Delta_R(p)\, (v_R(p) - v_Q(p))$.*

Downhill transfer carries memory from a writer to readers below it, and by this corollary it helps them. Carried uphill, the same memory costs the adoption shift times the reader's lead, and each upgrade adds to that lead by raising the reader while the store stays at the writer's level.

Corollary 3.6 (Item selection)**.** *Under the conditions of Proposition 3.4, suppose every approach-carrying item carries a_S and serving any one of them raises adoption. Then for each subset T of the store that contains an approach-carrying item, $\tau_R(p; T)$ has the sign of $G_R(p)$. Only subsets with no approach-carrying item, the empty subset among them, escape it.*

A selector decides how much of the predecessor's approach to serve; removing all approach-carrying items removes it.

Corollary 3.7 (Conditional adoption)**.** *Suppose the store arrives after the reader holds a checked kernel, and in each episode the reader replaces that kernel with the stored approach only when the stored approach scores higher on p. Then serving the store never replaces the reader's checked kernel with a slower one, whatever the sign of $G_R(p)$.*

Under late delivery, one such design, the reader sees the store only after it holds a checked kernel of its own and can compare the two before it submits.

Remark 3.8 (What verification certifies)**.** For an item i with source problem q_i, the evaluator certifies correctness at 10^{-4} on q_i and records the speedup $s_i = s(i, q_i)$. Proposition 3.4 concerns $G_R(p)$ on a target $p \neq q_i$, since items naming p are removed. The gap depends on the reader and the target; the certificate depends on the item and its source. No condition on an item's correctness and s_i fixes the sign of $G_R(p)$ for every reader and target. A gate that compares s_i with the reader's level on q_i measures the right reader on the source problem instead of the target.

In the experiments we read the approach from the submitted code: its backend, CUDA C++ through `load_inline` or Triton, and whether it is a token kernel. A *token kernel* is one whose own code takes under 5% of runtime while a library call does the rest. The store's approach is Triton, most often a library call followed by a fused Triton epilogue. Sections 4.3 to 4.6, 4.8 and 4.10 test the proposition, each corollary, and the remark.

4 A predecessor's store makes its successor worse on KernelBench

4.1 Setup

Tasks. We drew 130 of KernelBench's 250 problems [21], with reference modules vendored verbatim from commit 423217d9. A preregistered filter used reference properties alone, requiring feasibility on a 40 GB A100 at enlarged sizes within memory and eager-time bounds and outputs that are finite, non-degenerate and without an input-independent answer within tolerance. It retained ten held-back problems (six Level 2, four Level 3) and 120 other problems, 110 from the learning pool and 10 development problems the harness never streamed. The cross-vendor test of Section 4.7 uses 21 further problems disjoint from the head-to-head set and from the randomized test's split.

Readers and metrics. The readers are Claude Opus 5.5, Claude Opus 4.8, and, in Section 4.7, GPT-6 Astra, called as `claude-opus-5-5`, `claude-opus-4-8`, and `gpt-6-astra`. Opus 4.8 is the predecessor whose work built the store. Opus 5.5 is its successor and, without the store, the stronger model on these problems. We report mean capped speedup, log speedup $\log(1 + s)$, and fast$_p$, the fraction of episodes correct and faster than p times eager, each averaged over problems.

Designs and baselines. The baseline is the bare model, the same agent loop with no store (Table 1), and each harm contrast is a comparison inside a preregistered design. In the randomized test we switched the whole store on or off independently with probability $\frac{1}{2}$, ten episodes per problem for Opus 5.5 and four for Opus 4.8. The confirmation test ran five arms in shuffled order on problems held out from calibration, from which the store was partly built; that favors the store and makes the successor's loss there conservative. The head-to-head compared nine arms, six of them forms of the store, on ten held-back problems, and the calibration run tested component removal and three serving changes (item gating, framing and late delivery).

Validity controls. We preregistered every design before its first episode (Table 11). The evaluator runs in a sandbox the agent cannot reach, produces references with fresh seeds and weights, and had to pass two legitimate kernels and reject 21 attacks before any arm ran. Items that name the current problem are removed before serving, and no item names a held-back problem; the client and the effort, token and turn limits were identical across arms, and for Opus 5.5 in the randomized and confirmation tests no episode had an infrastructure fault. Reference timing was inflated, evenly across arms, when evaluations shared a GPU. Normalizing it moves the randomized estimate from -0.103 to -0.105. We recomputed the statistics of every design except the calibration run and the store-age test from the recorded episodes at the exact commits with independent analysis code. The local check misread CUDA kernels that declare `__launch_bounds__` as having no custom kernel. This cost 71 of 466 bare and 36 of 454 store Opus 5.5 episodes a turn and scored one bare submission zero, which weighs on the bare arm. In the randomized test the store's cut in output tokens is 25.9% with those episodes and 26.6% without them.

Table 1: **Every design holds the agent loop, the store, and the evaluator fixed and varies only the serving decision.** Every arm runs the long-running harness's kernel agent loop (released with internal identifiers renamed; system prompt in Appendix A.3) at the provider's service default for reasoning effort and sampling (adaptive thinking; no temperature or top-p set). Each episode has at most five turns of at most 32,000 output tokens per reply, each a local check or the submission, and ends within 2,400 s of wall clock with one protected evaluation on an A100-SXM4-40GB. All arms share identical caps. Reps counts episodes per problem over both arms for the randomized test and per problem and arm for the other designs. The technique-library ACE arm is a re-implementation with roles split across two models (Section 5).

Design	Readers	Problems	Arms	Reps	Episodes
Randomized	5.5, 4.8	92	whole store on/off, $p = \frac{1}{2}$ per episode	10; 4	1,288
Confirmation	5.5, 4.8	28	bare, show-all, self-gate, two fitted reader rules	4	1,120
Head-to-head	5.5, 4.8	10	bare + eight serving arms, six of them forms of the store	4	720
Calibration	5.5	37	bare + eight component, gating, framing, and timing arms	≤ 2	317
Store age	5.5, 4.8	10	frozen snapshots, hours 6–36	4; bare 10	per snapshot
Cross-vendor	5.5, 4.8, Astra	21	bare, single-pass store, four second passes (self-refinement, store after an attempt, audited store, information-free control)	3; 2; 2	882
Technique library	5.5	26	bare, ACE, hour-36 store, store or ACE + cards	3	720

Statistics. We average repetitions within a problem and paired differences over problems. Intervals are percentile intervals from a problem bootstrap: 90% for the store-age and technique-library tests, 95% with 2,000 resamples for the calibration run, and 95% with 10,000 resamples for every other test. Up/down and win:loss counts use $|d| > 0.05$ in the randomized and technique-library tests and the sign of the difference in the head-to-head, confirmation, and cross-vendor tests. The cross-experiment count of Section 4.10 uses a one-sided Fisher exact test over sources.

4.2 The store repairs the predecessor and slows the successor

Switching the whole store on or off at random in 1,288 episodes changed Opus 5.5's log speedup by -0.103 $[-0.163, -0.048]$ over 92 problems and Opus 4.8's by $+0.256$ $[+0.178, +0.337]$ over 77, the problems where Opus 4.8's four episodes fell in both arms (Figure 1a). A log-speedup difference of -0.103 is a ratio of 0.90 between the geometric means of $1 +$ speedup. The confirmation test on 28 problems outside calibration gave -0.182 $[-0.327, -0.074]$ for Opus 5.5 and $+0.218$ $[+0.149, +0.291]$ for Opus 4.8. Together, the two tests are the first randomized, replicated measurements of memory carried across a model upgrade, and the first to place the upgraded model below its level without memory.

The store cost Opus 5.5 more than 0.05 log speedup on 39 problems and helped it by that much on 14. For Opus 4.8 the counts reverse, 7 and 44. Confirmation repeated the pattern on 22 of its 28 problems for Opus 5.5 and 24 for Opus 4.8. On the same problems the same items carried opposite signs for the two readers, which ties the loss to the pairing of store and reader.

The two readers move on different axes (Table 2): the same items raise Opus 4.8's floor of correctness and lower Opus 5.5's ceiling of speed. For Opus 4.8 the store works as a repair: its failures fell from 62 of 186 episodes to 3 of 182, and all 46 of its compile errors without the store were CUDA episodes, against none with it. Opus 5.5 was correct in 918 of 920 randomized episodes. All of its loss is in speed, -0.105 $[-0.165, -0.051]$ log speedup among correct kernels, and its share of kernels at twice eager speed fell from 37.7% to 26.2%.

We tested eight alternatives that could lower the successor's mean without anchoring, and each fails a test the recorded episodes already contain (Table 3). In the most direct test, the same verbatim store, served after the reader's first check instead of in the opening prompt, scored $+0.333$ above up-front delivery, which shows that the characters alone do not carry the loss. The same rendered store on the same problems moved the two readers in opposite directions on 39 problems and the reverse way on 5, which rules out items that are bad in themselves.

4.3 A worked case: the successor writes its predecessor's kernel

KernelBench L2_074 applies a transposed 3D convolution from 16 to 32 channels, with kernel size 3 and stride 2, to a $16 \times 16 \times 16 \times 32 \times 32$ input. An epilogue of two LeakyReLUs around a per-channel multiply follows, then a

Table 2: **The store repairs Opus 4.8's correctness and cuts Opus 5.5's fast kernels.** fast_p is the fraction of episodes correct and faster than p times eager, averaged over problems. Differences are store minus no store, paired by problem, with 95% problem-bootstrap intervals. Randomized test: 92 problems for Opus 5.5, 77 for Opus 4.8; confirmation: 28 problems.

		Claude Opus 5.5			Claude Opus 4.8		
Design	Metric (↑)	off	on	difference	off	on	difference
Randomized	fast_0 (correct)	0.996	1.000	$+0.004$ $[+0.000, +0.010]$	0.673	0.968	$+0.294$ $[+0.203, +0.387]$
Randomized	fast_1	0.855	0.843	-0.012 $[-0.051, +0.025]$	0.357	0.604	$+0.247$ $[+0.145, +0.346]$
Randomized	fast_2	0.377	0.262	-0.115 $[-0.179, -0.054]$	0.056	0.117	$+0.061$ $[+0.015, +0.113]$
Randomized	fast_5	0.141	0.112	-0.030 $[-0.061, -0.003]$	0.026	0.017	-0.009 $[-0.026, +0.000]$
Confirmation	fast_1	0.893	0.723	-0.170 $[-0.304, -0.045]$	0.366	0.598	$+0.232$ $[+0.098, +0.366]$
Confirmation	fast_2	0.304	0.188	-0.116 $[-0.232, -0.018]$	0.009	0.036	$+0.027$ $[+0.000, +0.080]$

Table 3: **The loss survives each alternative explanation the episodes can test.** Each row states an alternative, the test that separates it from predecessor anchoring, and the result for Claude Opus 5.5. Δ is $\log(1 + \text{capped speedup})$ with 95% problem-bootstrap intervals unless marked: capped means mean capped speedup; 90% marks the store-age and technique-library tests.

Alternative	Test	Result
Context length	Identical store text at the first check instead of the opening prompt (calibration run)	late − up-front $+0.333$ $[+0.125, +0.519]$, 24 problems
	An information-free second pass in the same framing, 5,471 against 15,377 characters (cross-vendor test)	Opus 5.5 -0.275 $[-0.579, -0.019]$, GPT-6 Astra -0.359 $[-0.625, -0.140]$; capped, 21 problems
	Dose: ACE's 305,234-character playbook against the hour-36 store (6,982 characters on average), each minus bare	$+0.09$ $[-0.10, +0.32]$ against -0.14 $[-0.23, -0.05]$; capped, 90%, 26 problems
Bad items	The same rendered store, both readers, same problems (randomized)	opposite signs on 39 of 77 problems, reversed on 5; reader gap $+0.350$ $[+0.245, +0.462]$
Evaluator timing	Rescore the 920 Opus 5.5 episodes with references normalized	-0.105 $[-0.166, -0.050]$
The 10× cap	Uncapped speedup	-0.103 $[-0.163, -0.046]$
Regression to the mean	Strength terciles cross-fitted on disjoint halves, 119 problems	strongest − weakest tercile -0.278 $[-0.410, -0.140]$
A stale store	The hour-8 snapshot, 1.92 hours of predecessor work	-0.456 $[-0.875, -0.042]$; capped, 90%, 10 problems
Unrefereeable references	Drop L3_045 and L3_048, whose fp32 references fall outside tolerance against fp64 [8]	-0.098 $[-0.157, -0.041]$, 90 problems; Opus 4.8 $+0.254$ $[+0.174, +0.334]$
Off-distribution problems	Problems held out from calibration against calibration problems	-0.185 $[-0.342, -0.038]$ (27) against -0.069 $[-0.120, -0.022]$ (65)

$2 \times 2 \times 2$ max pool. The convolution's output has 67 million elements and the pool reduces it to 8.4 million. Eager PyTorch takes 5.9 ms. It writes the full output and passes over it four more times.

The successor alone. In all seven bare episodes Opus 5.5 wrote one CUDA kernel for the whole module (Figure 4a). The kernel exploits a structure of the problem: a stride-2 transposed convolution splits into eight parity classes, and each output in a $2 \times 2 \times 2$ pooling window receives a fixed subset of the 27 kernel taps from a small neighborhood of the input. Each thread holds the eight window outputs for two pooled positions and four output channels in registers, applies the bias, the epilogue and the maximum there, and writes only the pooled value, so the convolution's output never reaches memory.

The kernel ran in 0.61–0.91 ms. Its seven episodes scored 6.52–$9.68\times$, $8.86\times$ on average (Table 4). The successor spent 13,276 output tokens per episode on it, over two or three turns.

The successor with the store. With the store, Opus 5.5 read the 10,364 characters of Figure 3. In all three episodes it kept the convolution in cuDNN and wrote one Triton kernel for the epilogue and pool (Figure 4b), which it checked once and submitted. It wrote 2,958 output tokens per episode and scored 1.263–$1.268\times$.

Opus 4.8 wrote the same design. Its bare episodes kept the convolution in cuDNN, one with a Triton epilogue at $1.27\times$ and one with a CUDA epilogue that failed to compile. With the store it scored $1.26\times$ and $1.27\times$. On this problem the

Table 4: **On L2_074 the store makes the successor write its predecessor's design.** Randomized test, every episode on the problem. Own-kernel share is the fraction of runtime spent in the submission's own kernels; the rest is library code. Tokens are output tokens per episode.

Reader, store	Episodes	Design	Capped speedup	Own-kernel share	Tokens
Opus 5.5, off	7	one CUDA kernel for the module	6.52–9.68 (mean 8.86)	0.99–1.00	13,276
Opus 5.5, on	3	cuDNN + Triton epilogue	1.263–1.268	0.04–0.05	2,958
Opus 4.8, off	2	cuDNN + Triton or CUDA epilogue	1.274; compile error	0.04	1,854
Opus 4.8, on	2	cuDNN + Triton epilogue	1.260–1.271	0.05	2,304

store made the two readers indistinguishable by replacing the successor's kernel with the predecessor's.

The ceiling. In the episode of Figure 4b the Triton kernel took 5.1% of 4.66 ms, so the library call took 4.42 ms, and no epilogue the reader writes removes those 4.42 ms. The speedup of the adopted design over the 5.91 ms reference is at most $5.91/4.42 = 1.34\times$.

The bare kernel computed the entire module in 0.61 ms, 7.3 times faster than the library call the store keeps. The successor's slowest unaided episode, $6.52\times$, is 4.9 times the ceiling of the approach it adopted. To help the successor here, a stored approach must exceed $6.5\times$. The best recipe served was verified at $2.29\times$. On this problem the adoption shift of Proposition 3.4 is 1, from seven CUDA episodes to three cuDNN-plus-Triton episodes, and the store effect of -1.46 log speedup is the approach gap itself.

Two episodes, turn by turn. In bare episode 609 (h1-00609 in the released episode file), Opus 5.5 spent turn 1 writing a 6,197-character CUDA kernel (9,077 output tokens) and checked it. The local checker took the kernel's `__launch_bounds__` qualifier for its name and returned `no_custom_kernel`. Turn 2 rechecked the kernel (3,136 tokens) and found it correct, with a largest error of 4.2×10^{-7}. Turn 3 submitted it (2,874 tokens). The protected evaluator measured 0.609 ms against 5.864 ms for eager PyTorch, $9.63\times$, after 678 seconds. In store episode 814 (h1-00814 in the released episode file), Opus 5.5 spent turn 1 on a 2,197-character Triton kernel (1,920 tokens), checked it correct, and submitted it on turn 2 (1,133 tokens). The evaluator measured 4.658 ms against 5.907 ms, $1.27\times$, with own kernels at 5.1% of runtime, after 56 seconds. The local check reports no timing, and the reader with the store submitted its first correct kernel without ever seeing that it was slow.

The rebuilt recipe. The harness had removed the recipe for L2_074 itself, the record of Section 3.2, because it names the problem, and the successor wrote it anyway. Its kernel matches each clause of the removed recipe except the initial value of the maximum. It keeps the convolution in `nn.ConvTranspose3d` and fuses the rest into one Triton kernel that reads the convolution output once, with one lane per pooled output on a flat `BLOCK=256` grid and div/mod index decoding; inside, it loads the multiplier per channel and unrolls the window with `tl.static_range`, taking both slopes with `tl.where`. The rebuilt design scored $1.27\times$, within 0.01 of the removed recipe's $1.26\times$.

Removing one item left the approach in place, as Corollary 3.6 predicts, because the rest of the store carried it. Four of the five served recipes begin by keeping the transposed convolution in cuDNN, and the fifth rewrites it as a cuDNN `conv2d`. Both served kernels call the library convolution first, and two pitfalls warn against replacing it. The successor's index arithmetic follows the served L2_078 kernel line for line.

Gating and timing on the same problem. In the calibration run's first episodes we served L2_074 two more ways. Gating items by the reader's own level on their source problems served the $1.32\times$ recipe, which exceeds Opus 5.5's bare $1.26\times$ on its source problem L2_015, and the successor wrote a $1.26\times$ Triton kernel where it wrote $9.76\times$ alone. The gate measured the reader on the source problem, and the sign is set on the target (Remark 3.8). Delivered after its first check, the whole store left it at $10.40\times$. It checked its own CUDA kernel, read the store, and submitted its own code, as Corollary 3.7 describes.

4.4 The approach shift carries the loss

Across the randomized test the store moved Opus 5.5 as it did on L2_074. The successor's CUDA C++ share fell from 43.35% to 13.22% (Figure 5a). Token kernels rose from 19.7% to 31.9% of episodes. CUDA was its faster technology

(a) Bare Opus 5.5, episode 609, CUDA C++, $9.63\times$ (inner loop and epilogue; indentation compressed)

```
for (int ci = 0; ci < 16; ci++) {
  ...
  for (int kd = 0; kd < 3; kd++) {
    const int a = (kd == 1) ? 0 : 1;
    const int id = (kd == 0) ? 1 : 0;
    ...
          const int par = a * 4 + b * 2 + c;
          float4 wv = wc[(kd * 3 + kh) * 3 + kw];
          for (int j = 0; j < 2; j++) {
            float xv = xin[id][ih][iw + j];
            acc[j][par][0] = fmaf(wv.x, xv, acc[j][par][0]);
            ...
}
...
      for (int p = 0; p < 8; p++) {
        float v = acc[j][p][o] + bv;
        v = v > 0.f ? v : 0.2f * v;
        v = v * mv;
        v = v > 0.f ? v : 0.2f * v;
        mx = fmaxf(mx, v);
      }
      out[((long long)(n * 32 + oc) * D + pd) * PHW + ph * W + w0 + j] = mx;
```

(b) Opus 5.5 with the store, episode 814, Triton, $1.27\times$ (window loop and `forward`)

```
res = tl.full((BLOCK,), float("-inf"), tl.float32)
for dz in tl.static_range(2):
  iz = od * 2 + dz
  for dy in tl.static_range(2):
    iy = oh * 2 + dy
    rowbase = ((nc * D64 + iz) * H64 + iy) * W64 + ow * 2
    off = rowbase[:, None] + j[None, :]
    v = tl.load(x_ptr + off, mask=mask2, other=0.0)
    v = tl.where(v > 0, v, v * 0.2)
    v = v * mult2
    v = tl.where(v > 0, v, v * 0.2)
    res = tl.maximum(res, tl.max(v, axis=1))
...
def forward(self, x):
  y = self.conv_transpose(x).contiguous()
  ...
  BLOCK = 256
  grid = (triton.cdiv(total, BLOCK),)
  fused_pool_kernel[grid](y, m, out, C, D, H, W, OD, OH, OW, total, BLOCK=BLOCK, num_warps=4)
```

Figure 4: **Two kernels for one problem.** (a) Alone, the successor computes the transposed convolution itself: each thread accumulates the eight parity outputs of a pooling window, `acc[j][par][·]`, and writes only the pooled maximum. (b) With the store, it calls cuDNN for the convolution and writes a Triton kernel for the epilogue and pool, 5.1% of its runtime. Both kernels are verbatim; Appendix B.3 gives them in full.

here: within problems, its CUDA episodes were faster by $+0.17$ $[+0.06, +0.29]$ log speedup. Opus 4.8 moved the same way, from 47% CUDA to 1%, with the opposite sign, because the move removed all its compile errors.

Equation (2) predicts the size of the backend channel. The adoption shift away from CUDA is 0.301 and the gap is -0.17, so the adoption term through backend is $0.301 \times (-0.17) = -0.051$. We measured the same quantity independently with a problem fixed-effects regression, in which adding a CUDA indicator moves the store coefficient from -0.106 $[-0.168, -0.053]$ to -0.053, a step of 0.053. The prediction and the measurement agree to 0.002.

Adding the token-kernel indicator takes the coefficient to -0.024 $[-0.065, +0.015]$, an interval that includes zero; conditioning on backend and token-kernel status absorbs 77.6% of the store coefficient (Table 13). Within CUDA episodes the store's effect is $+0.034$ $[-0.094, +0.155]$, and within every kernel class of Table 5 but token kernels the store-served mean is at or above the bare mean, and we detect no within-approach loss where it can be read.

We classified every randomized episode by what the reader built (Table 5). Own CUDA C++ kernels, Opus 5.5's fastest class at $4.87\times$ on average, fell from 127 of 466 episodes to 43 of 454. Library-plus-Triton designs and token kernels rose from 132 to 232. The 25 store episodes that lost most against their problem's bare mean contain no own CUDA kernel, whereas 33 of the 48 bare episodes on those problems are own CUDA. For Opus 4.8 the same store emptied the incorrect class, 62 episodes to 3. All 46 compile errors went with it.

Token kernels carry the same mechanism inside Triton, and the ceiling of L2_074 recurs with them at a smaller scale.

Table 5: **The store moves the successor out of its fastest class and the predecessor out of its failures.** Every episode of the randomized test by what the submitted kernel is. Classes are exclusive and assigned in order: incorrect (compile, runtime, wrong output, or rejected); slower than eager; token kernel (own kernels under 5% of runtime); library plus epilogue (own kernels 5–50%); own kernel (50% or more), split by backend. Mean capped speedup is Opus 5.5's within the class.

	Claude Opus 5.5		Claude Opus 4.8		Opus 5.5 mean ×
Class	no store	store	no store	store	no store / store
incorrect	2	0	62	3	0 / –
slower than eager	64	77	61	70	0.62 / 0.65
token kernel	80	124	36	47	1.61 / 1.43
library + Triton epilogue	52	108	14	43	1.90 / 1.98
library + CUDA epilogue	28	3	7	1	1.92 / 4.28
own Triton	113	99	5	18	2.63 / 3.23
own CUDA C++	127	43	1	0	4.87 / 6.53
episodes	466	454	186	182	

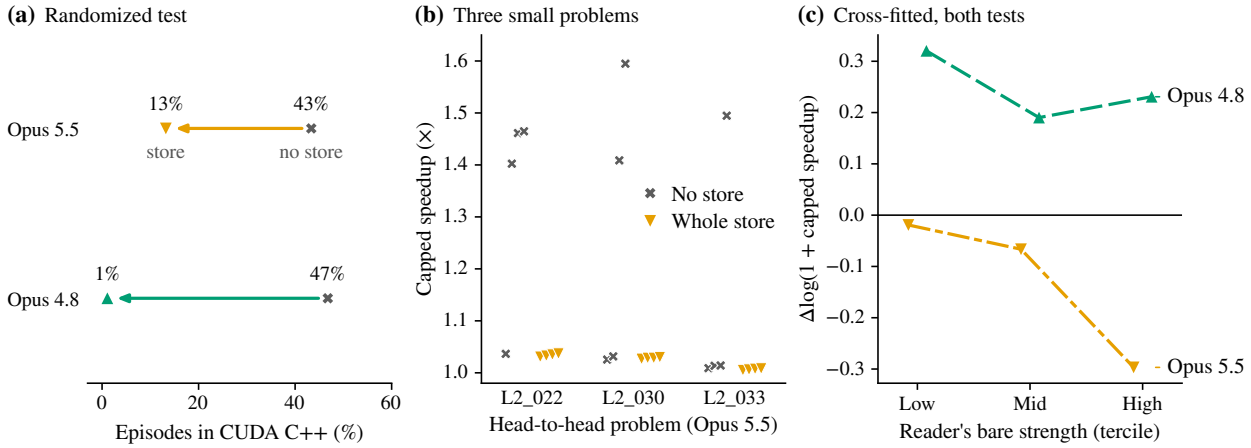

Figure 5: **The store moves Opus 5.5 off its faster kernels and costs it most on the problems where it is strongest.** (a) Share of episodes in CUDA C++ in the randomized test for each reader. Arrows run from no store to the store. (b) Opus 5.5 capped speedups on three small head-to-head problems, four episodes per arm, with no store (bare) and with the whole store. Every whole-store episode is a token kernel. (c) Store effect in log speedup by tercile of the reader's bare strength, cross-fitted over 200 random splits of both tests.

On three small head-to-head problems (L2_022, L2_030, L2_033), bare Opus 5.5 reached 1.40–$1.60\times$ in 6 of its 12 episodes, each time with a kernel that does the whole computation itself. With the whole store all twelve episodes were token kernels behind cuBLAS, and all scored 1.005–$1.037\times$ (Figure 5b).

Over the 120 problems of both tests, the store also shortened the search, cutting Opus 5.5's output from 9,597 to 7,142 tokens per episode, a paired cut of 2,588 tokens or 27% of the no-store level, and raising submissions right after a first check from 76% to 86% (Table 13). On L2_074 the cut was 78%. Conditioning on the shortened search and the backend shift absorbs the loss: on the same 1,256 Opus 5.5 episodes, a regression with problem fixed effects puts the store coefficient at -0.129 $[-0.185, -0.075]$, and adding the CUDA indicator, log output tokens, and turns takes it to -0.012 $[-0.044, +0.024]$.

Corollary 3.5 puts the loss at the adoption shift times the reader's lead, and it concentrates where the successor's own kernels sit furthest above the store. On the eight problems where the store cost Opus 5.5 most, its bare means ran from $2.31\times$ to $8.86\times$ and its store means from $0.87\times$ to $2.25\times$ (Table 6). On six of the eight, all bare episodes were CUDA C++.

We cross-fitted bare strength and store effect on disjoint halves of the no-store episodes, which keeps regression to the mean from making strong problems look harmed. Opus 5.5's effects by tercile were -0.019, -0.066, and -0.297 log

Table 6: **The loss sits where the successor's own kernels are far above the store.** The eight problems with the largest store effect on Opus 5.5 in the randomized test. Means are capped speedup. CUDA counts episodes written in CUDA C++; token counts episodes whose own kernels take under 5% of runtime. Recipes and kernels: how many the rendered store carried for the problem.

		bare		store			
Problem	Family	mean	CUDA	mean	CUDA / token	Δ log	Recipes, kernels
L2_074 ConvTranspose3d_LeakyReLU_...	conv_transpose	8.86	7/7	1.27	0/3, 1/3	-1.46	5, 2
L2_008 Conv3d_Divide_Max_...	conv	4.09	5/5	1.06	0/5, 5/5	-0.89	6, 2
L1_085 conv_depthwise_2D_...	conv	3.44	5/5	1.08	0/5, 0/5	-0.75	6, 2
L2_056 Matmul_Sigmoid_Sum	matmul	4.34	0/6	1.88	0/4, 3/4	-0.73	6, 2
L3_048 Mamba2ReturnY	attention	4.07	2/5	1.10	0/5, 2/5	-0.69	0, 0
L2_067 Conv2d_GELU_GlobalAvgPool	conv	2.31	5/5	0.87	3/5, 0/5	-0.67	6, 2
L2_071 Conv2d_Divide_LeakyReLU	conv	2.45	2/2	1.00	2/8, 0/8	-0.57	6, 2
L2_057 Conv2d_ReLU_HardSwish	conv	4.74	2/2	2.25	0/8, 0/8	-0.57	6, 2

speedup, a fifteenfold rise from its weakest problems to its strongest (Figure 5c). Opus 4.8 gains in each tercile ($+0.32$, $+0.19$, $+0.23$).

The stored recipes sit where Corollary 3.5 places them: below the successor and above the predecessor. Only 4 of the 22 speedup-tagged recipe entries exceed Opus 5.5's bare mean on the problem where each was verified. The median entry reached $1.65\times$ against Opus 5.5's $2.43\times$. Of the 20 entries with Opus 4.8 data, 18 exceed that reader's bare mean, with medians of $1.47\times$ for the entries and $1.01\times$ for Opus 4.8.

On development problem L2_058 the successor reached the $10\times$ cap alone and $7.2\times$ with the store's family recipes, which kept ConvTranspose3d in cuDNN and fused the epilogue, a $1.3\times$ strategy from Opus 4.8's solutions. Hiding those recipes restored $10\times$ in 8 of 8 runs.

4.5 Why verification does not prevent anchoring

Each kernel and recipe in the store passed an evaluator 100 times stricter than the audited KernelBench check, which made the store trustworthy, and the store still anchored the successor for the reason given in Remark 3.8. The certificate fixes an item's correctness and its speedup on the item's source problem, while the store's effect on a reader such as Opus 5.5 depends on that reader's level on each target problem, which the certificate does not record.

Verification certifies a floor. The evaluator certifies correctness and records speed, and it admits items at parity. The two related kernels served on L2_074 are verified at $1.00\times$ and $1.06\times$. Of the 22 recipe entries, 18 do not exceed Opus 5.5's own bare mean on their source problems. For Opus 4.8, a failing reader, these guaranteed-correct designs at the writer's level were repairs, and its failures fell from 62 of 186 episodes to 3 of 182.

Opus 5.5 works above that level, and for it the same designs were a ceiling: the store's library-plus-epilogue recipes lifted Opus 4.8 out of its CUDA compile errors and held the successor to $1.26\times$ on L2_074. Given a digest of the randomized outcomes, Opus 5.5 summarized them as "The store raises the floor and lowers the ceiling."

A certificate does not move from source to target. *Frontier gating* is the direct repair: it serves a recipe or kernel only if its verified speedup exceeds the reader's own bare mean on the item's source problem, and we tested it in the calibration run. For Opus 5.5 the gate kept 4 of 22 recipe entries and 7 of 47 kernel entries. For Opus 4.8 it kept 19 of 22 and 38 of 47. Gated this way the store still scored -0.118 $[-0.306, +0.045]$ against no memory. Gating also against the reader's predicted level on the target problem left -0.087 $[-0.312, +0.122]$.

On L2_074 the gate passed the $1.32\times$ recipe, which was above Opus 5.5 on its source and far below it on the target. Neither gate had the reader's measured level on the target, which only the reader's own attempt on that problem provides.

A verified lesson can forbid a technique. Opus 4.8's failures with plain TF32 matrix multiplies became a lesson, written by Opus 5.5, to always use exact fp32 `tl.dot`. The rule is correct, since exact fp32 passes the 10^{-4} check, but it also excludes the fp32-accurate `tf32x3` mode, and none of the harness store's 181 kernel items used that mode.

ACE's playbook, grown on the same stream, kept `tf32x3`. On five GEMM-heavy problems it lifted Opus 5.5 (L2_040: $1.56\times$ against $1.03\times$ bare).

The transposed-convolution pitfalls served on L2_074 are of the same kind. Each turns one of Opus 4.8's failed attempts into a rule for Opus 5.5. Four of the five record a predecessor attempt that ran slower than eager, and the fifth records one that crashed.

Tagging items with evidence deepens the loss. The annotated arm served each item tagged with its measured effect for the reader from randomized episodes, in the form "measured for ⟨reader⟩ on training problems: score x1.08 when shown, 136 randomised episodes". We used it to test whether a reader given the evidence would weigh the items itself. For Opus 5.5 it was the worst of the six serving methods, -0.610 $[-1.450, -0.100]$ in mean capped speedup at 11,848 characters, with tags that left the stored approach unchanged and added to its authority.

4.6 Item selection leaves the approach in the store

The strongest alternative to anchoring is a store with a few bad items; if the loss came from them, a better choice of items would remove it. In the preregistered head-to-head we tested whether any rule that chooses or annotates items returns the successor to its bare level, serving the store to both readers with six serving methods on ten held-back problems (Table 7), each of which changes which items the reader sees. Corollary 3.6 predicts that none changes the sign for Opus 5.5 while an approach-carrying item remains. All six left Opus 5.5 below bare and lifted Opus 4.8, with every Opus 4.8 interval above zero.

Show-all. This arm serves the whole store as the harness does, 8,885 characters. Opus 5.5 lost 0.287 $[0.089, 0.591]$ in mean capped speedup against its bare $3.309\times$. It fell on nine problems and tied the tenth at the cap.

Provenance. The harness's own rule hides a recipe from any reader that did not produce the solution it was written from. Each recipe came from an Opus 4.8 solution, so Opus 5.5 saw none, but it still saw the lessons, the pitfalls, and the verified kernels, all built from Opus 4.8's kernel work. The loss grew to 0.483. The verified kernels carry the approach in code; on L2_074, for example, the successor's index arithmetic follows the served L2_078 kernel. At hour 36 the same rule hid all 35 recipes from Opus 5.5, and the remaining 12 lessons, 76 pitfalls, and verified kernels still scored below bare (Section 5).

Trained selector. We fitted a ridge regression over randomized item exposure to estimate each item's effect for each reader, and the selector served the items with a positive estimate. The preregistration's mechanical rule chose it as the primary policy. It cost 0.266 and failed its no-harm floor in all five problem families. A split-plot test that served each item group whole or not at random traced the failure to a loss spread across the items, with no group's coefficient for Opus 5.5 above 0.032 in magnitude, while lessons alone helped Opus 4.8 by 0.132 ($z = 3.47$).

Opus selection. Opus 5.5 chose items for each reader and problem from the per-item randomized outcomes that trained the selector, given as tables or as a digest of them. The selections were short, 1,141 and 631 characters on average. They served nothing on 4 of the 10 problems. They still cost 0.347 and 0.359.

Evidence tags. The annotated arm of Section 4.5 cost 0.610, the largest loss of the six.

Zero-shot self-gating. Asked per problem whether the store would help it, Opus 5.5 chose to see it on 55 of 55 problems. On the 28 confirmation problems that policy scored -0.204 $[-0.361, -0.080]$ log speedup below no store. The question comes before the reader has written a kernel, when it has nothing to compare the store with, and the gap of Proposition 3.4 is not visible from the opening prompt.

Gating and framing. Frontier gating failed in the calibration run for the reason given in Remark 3.8. The same run tried two renderings that keep the store in the opening prompt (Table 8). The reader-relative rendering served the verbatim store under a header stating that its items were verified for an older, weaker model, that their speedups are floors rather than targets, and that CUDA C++ and Triton are equally available, with Opus 5.5's own bare level printed beside each item. It gained $+0.106$ $[+0.008, +0.211]$ over show-all with the store still in the opening prompt. The numberless rendering, which removes all speedup numbers, changed nothing ($+0.002$).

We located the loss by removal in the calibration run (Table 8). Lessons alone left Opus 5.5 at its no-memory level, and so did lessons with pitfalls. Adding the recipes and verified kernels cost -0.244 $[-0.450, -0.077]$. As Corollary 3.6

Table 7: **Each serving method leaves Opus 5.5 below bare and lifts Opus 4.8.** 10 held-back problems, 4 repetitions per cell. Differences from bare have individual 95% problem-bootstrap intervals; all arms share one bare cell. Bare means: Opus 5.5 $3.309\times$, Opus 4.8 $0.593\times$. Characters are means served to Opus 5.5. Up/down counts problems above or below bare in capped speedup.

		Claude Opus 5.5			Claude Opus 4.8
Arm	Characters (↓)	Δ capped (↑)	Δ log (↑)	up/down	Δ capped (↑)
show-all	8,885	-0.287 $[-0.591, -0.089]$	-0.069 $[-0.105, -0.034]$	0/9	$+0.216$ $[+0.071, +0.383]$
provenance	6,564	-0.483 $[-1.009, -0.092]$	-0.095 $[-0.155, -0.041]$	1/8	$+0.185$ $[+0.019, +0.370]$
selector (trained)	6,892	-0.266 $[-0.489, -0.085]$	-0.064 $[-0.098, -0.031]$	1/8	$+0.224$ $[+0.056, +0.439]$
Opus+digest	631	-0.359 $[-0.750, -0.068]$	-0.072 $[-0.117, -0.032]$	1/8	$+0.263$ $[+0.096, +0.441]$
Opus+tables	1,141	-0.347 $[-1.003, +0.232]$	-0.050 $[-0.142, +0.046]$	2/7	$+0.281$ $[+0.111, +0.461]$
annotated (tags)	11,848	-0.610 $[-1.450, -0.100]$	-0.121 $[-0.267, -0.023]$	2/8	$+0.271$ $[+0.110, +0.441]$

Table 8: **Removing the approach items removes most of the loss.** Calibration run: Opus 5.5 on 37 problems where its bare kernels are mostly CUDA C++, 317 episodes, up to two per arm and problem. Contrasts are paired by problem over the 17–25 problems each pair shares, with 95% problem-bootstrap intervals (2,000 draws); they are not additive.

Arm	What the reader saw	Contrast	Δ log speedup (↑)
show-all	the verbatim store in the opening prompt	− none	-0.272 $[-0.466, -0.081]$
lessons only	the 14 lessons	− none	$+0.001$ $[-0.066, +0.073]$
no approach	lessons and pitfalls, no recipes or kernels	− none	-0.001 $[-0.119, +0.115]$
show-all	recipes and kernels added to the above	− no approach	-0.244 $[-0.450, -0.077]$
no numbers	the store with all speedups removed	− show-all	$+0.002$
reader-relative	the store framed as an older model's floors, with the reader's own levels	− show-all	$+0.106$ $[+0.008, +0.211]$
frontier-gated	items whose speedup exceeds the reader on their source problem	− none	-0.118 $[-0.306, +0.045]$
target-gated	the above, also exceeding the reader's predicted level on the target	− none	-0.087 $[-0.312, +0.122]$
late	the verbatim store after the first local check	− none	$+0.106$ $[+0.008, +0.210]$
late	the same	− show-all	$+0.333$ $[+0.125, +0.519]$

predicts, the loss sits in the approach items and is shared among them. Removing the speedup numbers left it in place, as did hiding recipes by authorship or removing a single item. All six serving methods in the head-to-head left the successor below bare.

4.7 Across vendors, a second pass spent on the store falls below an information-free control

Corollary 3.5 makes no reference to the vendor, and it predicts that any reader above the store's level loses by adopting its approach and that the reader's own measured kernel is worth more to it than its predecessor's designs. We checked both predictions in a second preregistered test on 21 KernelBench problems that no other test, store item, or design file touches, with GPT-6 Astra as a third reader. Each second-pass arm starts from the reader's own bare kernel and its protected evaluation, and spends one more pass. In *self-refinement* the reader sees only its own result. *Consult* adds the store as an optional reference, with the instruction that the first solution stands unless beaten. The *information-free control* adds a fixed four-sentence diligence instruction in the same framing, with no store content (second-pass contexts of 5,471 and 15,377 characters on average for Opus 5.5). We keep the better of the two passes, and a fresh protected evaluation scores the final kernel, which removes the winner's-curse bias of keeping a maximum.

Spending the second pass on the store left both strong readers below the information-free control in mean capped speedup, Opus 5.5 by -0.275 $[-0.579, -0.019]$ and GPT-6 Astra by -0.359 $[-0.625, -0.140]$, with their own checked kernels already in hand and an instruction that each stood unless beaten. For Opus 4.8 the same contrast was $+0.055$.

A second pass on the reader's own measured kernel lifted every reader. Opus 5.5 gained $+0.855$ $[+0.500, +1.261]$, with 18 problems up and 1 down, GPT-6 Astra gained $+1.031$ $[+0.557, +1.551]$ (17 and 1). Opus 4.8 gained $+0.529$ $[+0.319, +0.793]$ (Figure 6). Against self-refinement at matched compute, the preregistered primary contrast, the store-consult pass trailed by -0.30 $[-0.82, +0.25]$ for Opus 5.5 and -0.28 $[-0.67, +0.05]$ for GPT-6 Astra.

The sign flip of the randomized test reappeared on these problems, which no store item was built from. Served in the

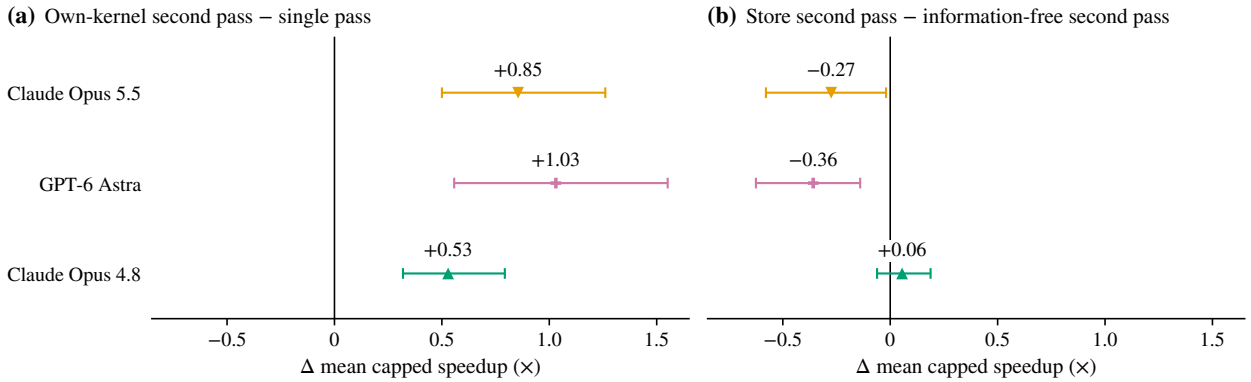

Figure 6: **Across vendors, the store makes a second pass worse than an information-free one for both strong readers, while the reader's own measurement lifts every reader.** Differences in mean capped speedup on 21 KernelBench problems that no other test or store item touches. Bars are 95% problem-bootstrap intervals. Both panels share one x axis. (a) A second pass on the reader's own measured kernel minus a single pass. (b) A second pass that consults the hour-24 store minus a second pass with a fixed four-sentence diligence instruction in the same framing. Each second-pass arm keeps the better of its two passes, and a fresh protected evaluation scores the kept kernel.

opening prompt, the store cost Opus 5.5 -0.773 $[-1.529, -0.191]$ and lifted Opus 4.8 by $+0.487$ $[+0.249, +0.716]$.

4.8 The remedies that work act on adoption

Under late delivery the store must compete with a kernel the reader has already checked, and in the ideal case of Corollary 3.7 a stored approach replaces that kernel only where it is better. We tested late delivery in the calibration run on 37 problems where bare Opus 5.5 writes mostly CUDA C++. The late arm left the opening prompt without memory and appended the verbatim store to the feedback on the reader's first local check, under the header "Memory from earlier problems (verified alternatives from other problems; keep your own code if you judge it faster)".

Late delivery scored $+0.333$ $[+0.125, +0.519]$ log speedup above up-front delivery (24 problems) and lifted Opus 5.5 above no memory by $+0.106$ $[+0.008, +0.210]$ (20 problems). Up-front delivery cost -0.272 $[-0.466, -0.081]$ (25 problems). The CUDA share was 0.89 without memory, 0.15 with the store up front, and 0.91 with it late. Both arms served the same store items, and the late arm also added the instruction to keep the reader's own code if judged faster; the reader supplied its own comparison, and the remedy used no model of the reader.

The second remedy withholds the store where its sign is negative; Proposition 3.4 makes the sign a property of a reader and a target, and the randomized test shows it is nearly a property of the reader. Given a digest of the randomized outcomes, Opus 5.5 chose to serve the store to Opus 4.8 and mostly withhold it from itself (serve rates 1.0 and 0.29). Frozen before confirmation, this rule kept Opus 4.8's gain ($+0.204$ $[+0.131, +0.282]$ over no store) and lifted Opus 5.5 by $+0.135$ $[+0.038, +0.250]$ over show-all, with 19 problems up and 9 down, meeting both preregistered predictions. A ridge predictor fitted to the same outcomes served Opus 5.5 on 7% of problems and came within 0.003 of no store. It kept Opus 4.8's gain of $+0.204$.

Choosing problem by problem would gain little over this rule: on the 27 held-out problems of the randomized test, an in-sample oracle that served Opus 5.5 the store on exactly the problems where it helped would have served 22% of them and gained 0.042 over withholding it everywhere. A fitted per-problem harm predictor separated helped from harmed problems at chance (AUC 0.50). For the successor, the useful content of the store came down to one bit, whether to withhold it. Randomized exposure revealed that bit, and the zero-shot prompt that asked Opus 5.5 the same question returned the opposite bit on all 55 problems.

In a preregistered probe outside kernels we tested a third control on adoption, the authority an item is given. On 16 synthetic API-call tasks, an item carried over from earlier runs contradicted the task's own documentation. Marked mandatory, it was followed in 16 of 16 tasks by Claude Opus 5.5 and by Claude Haiku 4.5. The preregistered prediction for Opus 5.5 was 30–70%. Marked advisory, it was followed in none. Opus 5.5 still took the benefit of a correct carried-over item in 16 of 16 (one-sided Fisher exact $p = 1.7 \times 10^{-9}$). In the kernel experiments, the evidence tags of Section 4.5 raised the store's authority and produced the largest loss of the six serving methods.

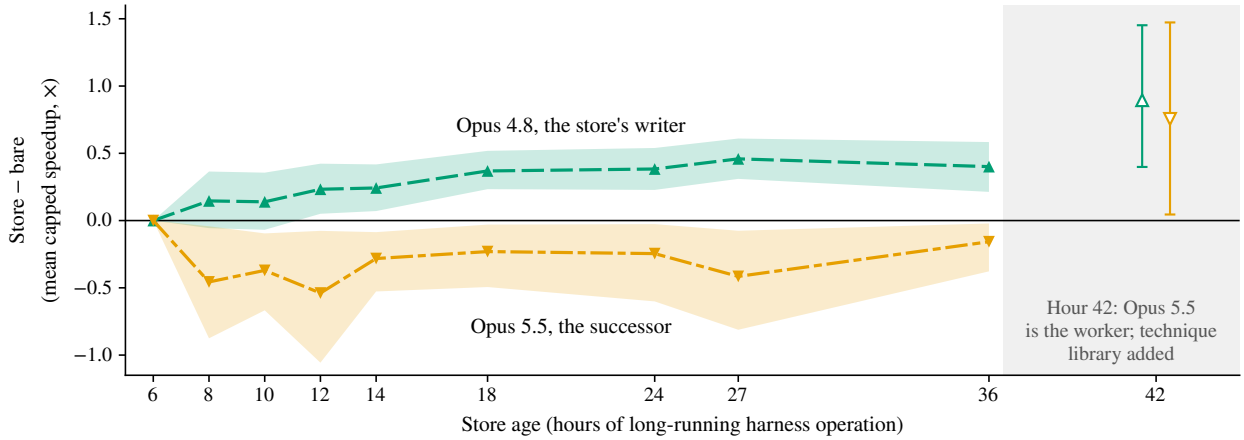

Figure 7: **Each snapshot from hour 8 to hour 36 slows the successor and lifts the predecessor.** Frozen snapshots minus bare in mean capped speedup on ten development problems, for the successor and for the predecessor that wrote the store; store cells $k = 4$, bare $k = 10$. Bands and bars are 90% task-bootstrap intervals over the ten problems. Bare means: Opus 5.5 $2.35\times$, Opus 4.8 $0.78\times$. Kernel work began at hour 6.1. The first snapshot is empty, and its difference is zero. The hollow hour-42 points in the shaded window are the store after the upgrade, with the technique library added.

Table 9: **The sign of transfer follows the strength gap.** All recorded comparisons of carried knowledge against the same reader without it, across our experiments, by the gap g between reader and writer. Nonzero comparisons only; significant at the comparison's own test.

Direction	g	Comparisons	Helped	Significant help	Significant harm
downhill (writer stronger)	< 0	6	5	2	0
same model	0	5	3	0	1
uphill (reader stronger)	> 0	9	0	0	4

4.9 Under two hours of predecessor work anchors the successor

We served frozen snapshots of the store from hour 8 to hour 36 on ten development problems, and each lowered Opus 5.5's mean capped speedup, by 0.16 to 0.54, with all 90% intervals below zero (Figure 7). Kernel work began at hour 6.1. The hour-8 store, built from under two hours of predecessor kernel work, already anchored the successor. At hour 36 the store was still below bare by -0.16 $[-0.38, -0.02]$.

Opus 4.8's gain grew with the store, by $+0.32$ per day $[+0.10, +0.56]$ from hour 12 to hour 27, and reached $+0.46$ $[+0.31, +0.61]$, while the successor's loss did not grow. By hour 42, after the upgrade and with the technique library in the store, the store lifted both readers above bare: Opus 5.5 by $+0.76$ $[+0.05, +1.47]$ and Opus 4.8 by $+0.90$ $[+0.40, +1.45]$. All intervals in this subsection are 90%.

4.10 Downhill and uphill across our experiments

Corollary 3.5 is not specific to this store, and it predicts that wherever approach-carrying memory crosses models, the sign of transfer follows the strength gap between writer and reader. We rank the models in our experiments by strength: Claude Haiku 4.5 and Qwen3.5-27B at 1, Opus 4.8 at 2, Opus 5.5 at 3. The gap of a comparison is $g = \text{rank}(R) - \text{rank}(Q)$. We count a comparison when a reader is given knowledge distilled from a writer's past episodes, in any form from a store or playbook to written rules, recipes or checks, and is compared with the same reader without it; mid-task handoffs are excluded.

The recorded comparisons follow the corollary (Table 9). Downhill, where the stored approach sits above the reader, transfer helped in 5 of 6 comparisons; uphill, where it sits below, transfer helped in 0 of 9, with 4 significant harms. At equal strength the gap is near zero and the sign is mixed, 3 of 5. Over all 21 recorded comparisons, including the one tie that Table 9 omits, the sign falls with the gap (Spearman -0.76). The equal-strength row is the peer control, in which

stores read by their own writer helped in 3 of 5 comparisons, and against it the harm goes with the uphill direction of the gap. By source, downhill transfer helped in 3 of 3 sources and uphill transfer in 0 of 3 (Fisher one-sided $p = 0.05$).

Besides KernelBench, the pattern holds on AppWorld and on a long-horizon task in which an agent builds and operates an account-balance service, for stores and for written checks. An AppWorld store compiled by Opus 5.5 raised Claude Haiku 4.5 by $+7.0$ $[+2.9, +11.3]$ points. On the account-balance service, an evidence-only takeover harness carrying checks written by Opus 5.5 raised a Qwen3.5-27B successor by $+0.476$ $[+0.149, +0.803]$ over six streams. The kernel store is the controlled case of the uphill row, and reading the sign required only the ranks of writer and reader.

5 Verified technique transfers where experience does not

By Corollary 3.5, an item whose approach sits above the reader has a positive gap for any reader below it, whoever wrote the item, and a technique card is built to be such an item. The seed card carries the bf16 split of Henry, Tang, and Heinecke and of Ootomo and Yokota, which the successor did not use unaided: on the 42 episodes of problems where the seed card routes, bare Opus 5.5 used bf16 arithmetic in none, and with the library it did in 37, so on those problems the adoption shift moves the reader toward a better approach.

5.1 Setup

Tasks. We used 26 problems outside the card-search trials: 10 development problems the long-running harness never streamed, and 16 learning problems with no card-search trials and with same-problem solutions withheld.

Arms and baselines. We ran ten arms with Claude Opus 5.5 at three repetitions (720 episodes in total). The baselines are bare and ACE [34]. ACE is a run on the same stream to hour 36, with Opus 4.8 as generator and Opus 5.5 as reflector and curator. Its arm serves its own hour-36 playbook. For the ACE arm we re-implemented the generator, reflector, and curator loop of the ACE paper and repository (`https://github.com/ace-agent/ace`) in the long-running harness. The port adapts the prompts to kernels and runs one reflection round per episode. Its curator can also update, delete, and merge items, and it splits the roles across two models where the ACE paper uses one model in all three. We did not reproduce ACE's published AppWorld numbers. Other arms add the seed card or the library's text to ACE, or serve the long-running harness's hour-36 store alone or with all or part of the library. We added two arms by an amendment committed before their episodes. Intervals are 90% task-bootstrap intervals, and win:loss counts use $|d| > 0.05$.

5.2 Technique above the reader turns the sign positive

With the technique library, the harness's hour-36 store reached a mean capped speedup of $3.67\times$ for Opus 5.5, against $1.95\times$ for ACE's own hour-36 playbook and $1.86\times$ bare (Table 10). The store with the library exceeded ACE by $+1.71$ $[+1.19, +2.25]$, winning 22 problems and losing 4. On the 16 problems no card-search trial used, the margin was $+1.68$ $[+1.10, +2.31]$. Given the same literature seed, ACE still trailed by $+0.83$ $[+0.33, +1.40]$. Leaving out any one problem keeps the margin over ACE between $+1.55$ and $+1.85$.

Alone, the hour-36 store sat below bare, -0.14 $[-0.23, -0.05]$ with 6 problems up and 13 down. Its provenance rule had hidden all 35 recipes from Opus 5.5. The lessons, pitfalls, and verified kernels of the Opus 4.8 era still anchored it. With the library the same store gained $+1.94$ $[+1.47, +2.42]$, winning 24 problems and losing 1, and the two arms differ only by the six cards, so the library reversed the sign of the store itself.

The four cells of ACE's playbook and the hour-36 store, each with and without the six cards, form a factorial that separates the cards' share of the gain from the store's (Table 10). Without cards the store sits below ACE's playbook, -0.23 $[-0.43, -0.07]$, the same anchoring as before. The cards lift both, by $+1.94$ on the store and $+1.34$ on ACE, and they are worth $+0.60$ $[+0.16, +1.10]$ more inside the compact store than inside ACE's playbook, so the gain comes from the cards and they pay most in a short store. Dropping L3_048, whose reference fails its own fp64 check [8], moves the margin over ACE to $+1.85$ $[+1.35, +2.37]$ on 25 problems.

Added to ACE's playbook, a different memory, the library's text improved ACE by $+1.34$ $[+0.77, +1.91]$ (Figure 8). ACE's playbook filled 305,234 characters of context per episode, and the store with the library 18,478, 6% as much; over all turns the reader took in 290,871 input tokens per episode with ACE and 30,073 with the store and library (Table 10). ACE's playbook scored $+0.09$ $[-0.10, +0.32]$ over bare. The store with the library scored $+1.81$.

Table 10: **Verified technique lifts the successor past ACE's playbook with 10% of its input tokens.** Mean capped speedup; Opus 5.5, 26 problems, 3 repetitions. Paired contrasts have 90% task-bootstrap intervals; win:loss counts use $|d| > 0.05$. The 16 holdout problems had no card-search trials. Input tokens are the reader's, per episode, over all turns. Bold: best; underline: second. †Recomputed from the episode rows with the test's own estimator after dropping L3_048 Mamba2ReturnY, whose fp32 reference falls outside the tolerance against its own fp64 evaluation [8], leaving 25 problems.

Arm	Pooled (26) ↑	Dev (10) ↑	Holdout (16) ↑	Input tokens ↓
bare	1.86	2.14	1.69	8,378
ACE (own hour-36 playbook)	1.95	2.52	1.60	290,871
hour-36 store (provenance-filtered)	1.73	2.10	1.49	13,731
ACE + literature seed card	2.83	<u>3.72</u>	2.28	306,265
ACE + six cards (text)	<u>3.29</u>	3.68	<u>3.05</u>	312,458
hour-36 store + six cards	**3.67**	**4.29**	**3.28**	30,073

Contrast (↑)	Pooled (26)	W:L	Holdout (16)
(store + cards) − ACE	$+1.71$ $[+1.19, +2.25]$	22:4	$+1.68$ $[+1.10, +2.31]$
(store + cards) − (ACE + seed)	$+0.83$ $[+0.33, +1.40]$	17:7	$+0.99$ $[+0.34, +1.71]$
(ACE + cards) − ACE	$+1.34$ $[+0.77, +1.91]$	17:6	
(store + cards) − (ACE + cards)	$+0.38$ $[-0.03, +0.84]$	13:11	
store − ACE (no cards)	-0.23 $[-0.43, -0.07]$	2:18	
cards on store − cards on ACE	$+0.60$ $[+0.16, +1.10]$	18:8	
(store + cards) − ACE, without L3_048†	$+1.85$ $[+1.35, +2.37]$	22:3	
(store + cards) − bare	$+1.81$ $[+1.33, +2.28]$	22:3	$+1.58$ $[+1.06, +2.15]$
(store + cards) − store	$+1.94$ $[+1.47, +2.42]$	24:1	$+1.78$ $[+1.23, +2.41]$

On 12 convolution, transposed-convolution, and recurrent problems where the seed card does not apply, the search's cards added $+1.36$ $[+0.79, +2.10]$ to the store alone. They won 11 problems and lost none. One of them computes a convolution as an implicit GEMM with the bf16 split in registers, which moves the convolution's arithmetic from the 19.5 TFLOP/s fp32 path of the A100 to bf16 tensor cores that peak at 312 TFLOP/s. An earlier preregistered test on 12 further problems unused in card search found that the cards in a frozen port of the harness exceeded bare Opus 5.5 on all 12 ($+2.74$ $[+1.60, +3.96]$, 90%). We deployed the library into the running harness at hour 36, and it lifted the live system above bare by $+0.70$ in mean capped speedup on its probe problems.

The seed card sits above both readers and lifted both, Opus 4.8 from $0.75\times$ to $1.83\times$ on ten development problems ($+1.08$ $[+0.44, +1.74]$, 90%). A recipe records the writer's solution at the writer's level, and only 4 of the 22 recipe entries here exceeded the successor's bare mean on their source problems. A card records a technique and its envelope, and its level is set by the arithmetic it exploits, which at this upgrade lay above the successor's.

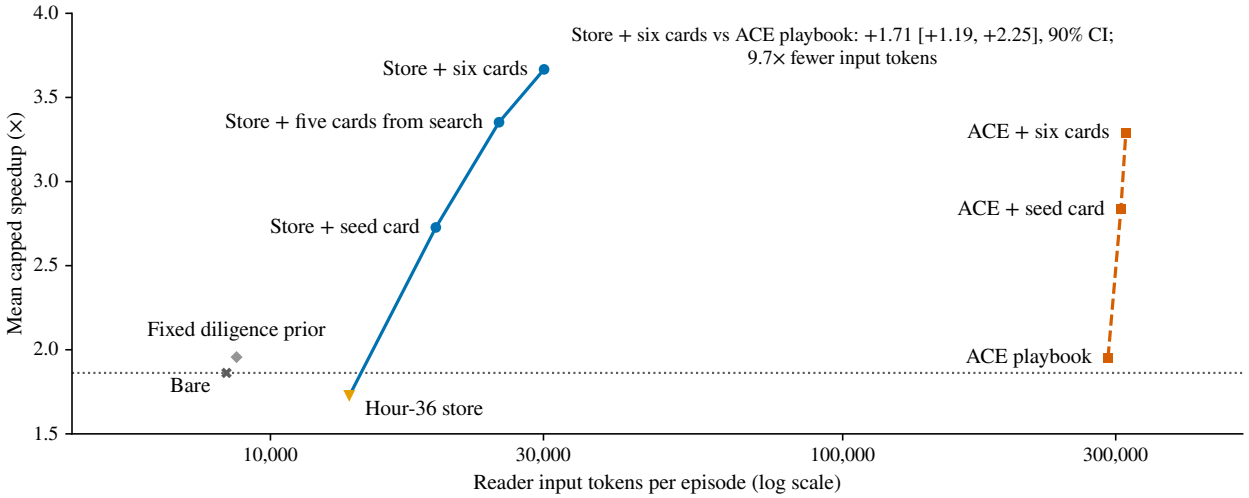

Figure 8: **Technique cards reach the highest speedup at a tenth of ACE's input.** Claude Opus 5.5's mean capped speedup against the reader's input tokens per episode over all turns (log scale), 26 KernelBench problems outside the card-search trials, 3 repetitions per arm. The solid line joins the hour-36 store alone and with one, five, or six evaluator-checked technique cards; the dashed line joins ACE's own hour-36 playbook alone and with the seed card or all six cards. Bare and a fixed diligence prior stand alone, and the dotted line is the bare level. The annotated contrast is paired by problem with a 90% task-bootstrap interval.

6 Conclusion

The store studied here holds approaches at its predecessor's level, each certified on the problem where it was found. For the predecessor the certified designs repaired its failures, and for the successor, which works above that level, they were a ceiling; on L2_074 the ceiling was $1.34\times$ under a reader that writes $8.86\times$ alone. The randomized effects and the backend decomposition measure the gap between the stored approach and the reader's own within one store, and the cross-vendor test and the cross-experiment sign pattern measure it across readers and sources; they agree in sign wherever the gap can be read.

The head-to-head's item rules and the successor's own judgment of the store all left Opus 5.5 below bare, and every approach-carrying item they kept carried the same approach. Late delivery acts on adoption by turning it into a comparison the reader makes with its own checked kernel in hand, and a reader-level rule acts on it by withholding the store where the gap is negative. Technique cards change the gap itself by carrying an approach above the reader's repertoire.

The next experiment varies the writer–reader gap inside one design, with stores built on one KernelBench stream by Claude Haiku 4.5, Opus 4.8, and Opus 5.5 as workers and frozen at matched hours. It serves each store to all three readers under the randomized on/off design and holds the agent loop and the evaluation, including the tolerance and the split, fixed. For all nine writer–reader pairs it measures the store effect, the adoption shift, and the approach gap of Proposition 3.4, and it randomizes the delivery point and the approach-carrying share of the store.

Each upgrade raises the reader and leaves the store at the old level, and the gap that drives predecessor anchoring widens with every generation a harness survives. A harness that carries memory across generations must measure each stored approach against its new reader before serving it, and what it carries upward must be technique that sits above that reader.

Acknowledgments

We thank the KernelBench maintainers, the KernelBench-M authors for their mutation suite, and the authors of the error-compensated bf16 methods used in the seed technique card.

References

[1] Zhengye Bao, Shengtao Zhang, Haokai Zhang, Shiqi Lou, Muning Wen, and Minnan Luo. Is Complex Agentic Memory Always Needed? Naive RAG Is Stronger Than We Thought. *arXiv preprint arXiv:2610.30755*, 2026. URL https://arxiv.org/abs/2610.30755.

[2] Merim Bilalić, Peter McLeod, and Fernand Gobet. Why good thoughts block better ones: The mechanism of the pernicious Einstellung (set) effect. *Cognition*, 108(3):652–661, 2008. doi: 10.1016/j.cognition.2008.05.005.

[3] Merim Bilalić, Peter McLeod, and Fernand Gobet. Inflexibility of experts—Reality or myth? Quantifying the Einstellung effect in chess masters. *Cognitive Psychology*, 56(2):73–102, 2008. doi: 10.1016/j.cogpsych.2007.02.001.

[4] Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, Ilya Sutskever, and Jeff Wu. Weak-to-Strong Generalization: Eliciting Strong Capabilities With Weak Supervision. *arXiv preprint arXiv:2312.09390*, 2023.

[5] Xueyi Chen, Shiyu Liu, Xin Jin, Yuhua Zheng, Xin Li, Haolei Bai, Junhan Zhu, and Huan Wang. MetaKernelBench: Measuring GPU Kernel Knowledge Transfer Beyond Code. *arXiv preprint arXiv:2610.05014*, 2026. URL https://arxiv.org/abs/2610.05014.

[6] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. *arXiv preprint arXiv:2504.19413*, 2025.

[7] DeepSeek-AI. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948*, 2025.

[8] Mingzhe Du, Anh Tuan Luu, Dong Huang, and See-Kiong Ng. Measuring the Checker: Mutation Analysis for GPU-Kernel Benchmark Oracles. *arXiv preprint arXiv:2609.22220*, 2026. KernelBench-M.

[9] Xuan He, Sining Zhoubian, Ruozhen Yang, Yuhao Wen, Zhineng Wang, Yuyang Hu, Ke Yang, Yuyu Zhang, Ping Nie, Jiawei Han, and ChengXiang Zhai. Memory Tax: Dissecting the Net Value of Agent Memory. *arXiv preprint arXiv:2610.34073*, 2026. URL https://arxiv.org/abs/2610.34073.

[10] Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke. Leveraging the bfloat16 Artificial Intelligence Datatype For Higher-Precision Computations. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 69–76, 2019. doi: 10.1109/ARITH.2019.00019.

[11] Prithwish Jana, Mononito Goswami, Hao Liu, Xinyu Li, Langlin Huang, Zhehui Huang, Zhishen Huang, Patrick Blöbaum, Anoop Deoras, Purak Jain, Nikos Kanakaris, and Sahika Genc. MILO: Automated Harness Discovery via Orchestrated Multi-Agent Evolution. *arXiv preprint arXiv:2609.38349*, 2026. Project page: https://jprithwish.github.io/MILO/.

[12] Changxiu Ji, Amy Lu, Qizheng Zhang, and Kunle Olukotun. Sentry: Learning to Recover from LLM Agent Failures at Test Time. *arXiv preprint arXiv:2610.02994*, 2026. URL https://arxiv.org/abs/2610.02994.

[13] Erik Jones and Jacob Steinhardt. Capturing Failures of Large Language Models via Human Cognitive Biases. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022.

[14] Slava Kalyuga, Paul Ayres, Paul Chandler, and John Sweller. The Expertise Reversal Effect. *Educational Psychologist*, 38(1):23–31, 2003. doi: 10.1207/S15326985EP3801_4.

[15] Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards Robust Agentic CUDA Kernel Benchmarking, Verification, and Optimization. *arXiv preprint arXiv:2509.14279*, 2025.

[16] Abraham S. Luchins. Mechanization in Problem Solving: The Effect of Einstellung. *Psychological Monographs*, 54(6):i–95, 1942. doi: 10.1037/h0093502.

[17] Yingzi Ma, Yulong Cao, Nanxi Li, Fang Wu, Yizhe Zhang, James Zou, Yejin Choi, and Chaowei Xiao. HarnessTTT: Learn to Optimize Harness at Test-Time for Scientific Discovery. *arXiv preprint arXiv:2610.17370*, 2026.

[18] METR. Measuring Automated Kernel Engineering. https://metr.org/blog/2025-02-14-measuring-automated-kernel-engineering/, 2025. Blog post, 14 February 2025.

[19] Saeid Alavi Naeini, Raeid Saqur, Mozhgan Saeidi, John Giorgi, and Babak Taati. Large Language Models are Fixated by Red Herrings: Exploring Creative Problem Solving and Einstellung Effect using the Only Connect Wall Dataset. In *Advances in Neural Information Processing Systems 36 (Datasets and Benchmarks Track)*, 2023.

[20] Hiroyuki Ootomo and Rio Yokota. Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance. *The International Journal of High Performance Computing Applications*, 36 (4):475–491, 2022. doi: 10.1177/10943420221090256.

[21] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs Write Efficient GPU Kernels? *arXiv preprint arXiv:2502.10517*, 2025.

[22] Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. *arXiv preprint arXiv:2509.25140*, 2025. ICLR 2026.

[23] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as Operating Systems. *arXiv preprint arXiv:2310.08560*, 2023.

[24] Hoang Pham, Cong-Duy Nguyen, Dong Le, Ponhvoan Srey, and Anh Tuan Luu. Reliable Memory from Unstable Growth: Curating Self-Written Agent Playbooks with Randomized Experiments. *arXiv preprint arXiv:2610.18427*, 2026.

[25] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. Large Language Models Can Be Easily Distracted by Irrelevant Context. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31210–31227. PMLR, 2023.

[26] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. *arXiv preprint arXiv:2303.11366*, 2023.

[27] Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic Cheatsheet: Test-Time Learning with Adaptive Memory. *arXiv preprint arXiv:2504.07952*, 2025.

[28] Shangyin Tan, Ben Zhou, Lakshya A Agrawal, Jialin Zhang, Yilin Chen, Qiuyang Mang, Donghyun Lee, Dennis Jacob, Koushik Sen, Alex Dimakis, and Matei Zaharia. Scaling Discovery Requires Forgetting. *arXiv preprint arXiv:2610.20563*, 2026. URL https://arxiv.org/abs/2610.20563.

[29] Philippe Tillet, H. T. Kung, and David Cox. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, pages 10–19. ACM, 2019. doi: 10.1145/3315508.3329973.

[30] 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. *arXiv preprint arXiv:2305.16291*, 2023.

[31] Yanze Wang, Pengfei Yao, Tianyi Sun, Hongda Chen, Yunyun Han, Jun Sun, Chuanrui Hu, and Yafeng Deng. Beyond Gain: Measuring Skill Internalization for Agent Skill Retirement. *arXiv preprint arXiv:2610.32322*, 2026. URL https://arxiv.org/abs/2610.32322.

[32] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. *arXiv preprint arXiv:2409.07429*, 2024.

[33] Zhihao Yang, Haojin Wang, Juncheng Wu, Kuan Zhang, Yukun Chen, Run Luo, Yaojie Lu, Yuyin Zhou, Lu Wang, and Xianpei Han. Evolving to Match Capabilities: Improving Harness Self-Evolution via Skill Adaptation. *arXiv preprint arXiv:2610.33024*, 2026. URL https://arxiv.org/abs/2610.33024.

[34] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. *arXiv preprint arXiv:2510.04618*, 2025. ICLR 2026.

[35] Yunxiang Zhang, Ping Yu, Jianyu Wang, Max (Xiangjun) Fan, Julian Reed, Azalia Mirhoseini, and Will Su. KernelBench-Verified: Do LLM-Generated Kernels Actually Beat PyTorch? *arXiv preprint arXiv:2607.16241*, 2026. URL https://arxiv.org/abs/2607.16241.

[36] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM Agents Are Experiential Learners. *arXiv preprint arXiv:2308.10144*, 2023. AAAI 2024.

[37] Jianghong Zhou, Hao Rong, Chen Lin, and Daoan Zhang. Many Steps into One: Reasoning Contraction and the Limits of Safe Memory Reuse for LLM Agents. *arXiv preprint arXiv:2610.17544*, 2026.

[38] Jiawei Zhou, Bo Peng, Yuxuan Huang, Ming Zhang, Youbang Sun, Xiang Wang, and Chaochao Lu. Does Your Generalist Skill Optimizer Really Incentivize Agentic Tasks? Rethinking Skill Optimization via Bipartite Attribution. *arXiv preprint arXiv:2610.37201*, 2026. URL https://arxiv.org/abs/2610.37201.

A Experimental details

We release result tables and verifiers at https://qwntl.com/paper-predecessor-anchoring.

A.1 Designs and preregistrations

Table 11: **Experimental designs and preregistrations.** All episodes used the same agent loop and evaluator. X: statistics recomputed independently from the recorded episodes, without rerunning them. Preregistrations are commits in our research repository; times are UTC on October 9, 2026.

Design	Randomized	Problems	Episodes	Readers	Check	Preregistration (time UTC)
Randomized	whole store on/off	92	1,288	5.5, 4.8	X	plan (06:23:43); run from 06:24:02
Confirmation	5 arms, 3 gate arms	28	1,120	5.5, 4.8	X	amendment, before any confirmation episode
Head-to-head	9 arms, 6 of them forms of the store	10	720	5.5, 4.8	X	plan + amendment; run from 05:08
Store age	frozen snapshots	10	per snapshot	5.5, 4.8		amendments to the store-age preregistration
Technique library	10 arms	26	720	5.5	X	plan + amendment
Cross-vendor	6 arms per block	21	882	5.5, 4.8, Astra	X	plan (06:58:40) + 5 amendments; Astra from 08:04
Calibration	9 arms	37	317	5.5		plan; run from 13:23

A.2 Estimation and checks

We bootstrap problems with 10,000 resamples for the randomized, confirmation, and head-to-head tests. Technique-library and store-age intervals are 90%; the others are 95%. Up/down counts use $|d| > 0.05$ in the randomized and technique-library tests, and the sign of the difference in the head-to-head and confirmation tests. Technology regressions use problem fixed effects with 2,000 bootstrap draws. Strength terciles use 200 random splits. The calibration run uses its own script with 2,000 draws and treats 17 evaluations that ran out of GPU memory on contended endpoints as infrastructure faults. The cross-experiment count uses a one-sided Fisher exact test over sources. Each technique-library result carries the evaluator's fingerprint, `fa88aae56eab8b0e`, computed from the evaluator files as run; the released files, with internal identifiers renamed and host paths removed, compute `58de1a4d04c4bb23`.

An independent recomputation from the recorded episodes at the exact commits reproduced every primary number of the randomized and confirmation tests and of the head-to-head and technique-library tests, including all 720 rows and the contrasts of the technique-library test, and rebuilt the randomized technology analysis the same way. For Opus 5.5 in the randomized and confirmation tests, correctness and truncation did not differ between arms, and neither did turns without code or forced unchecked submissions; for that reader in those tests no episode had an infrastructure fault, and the client and the effort, token and turn limits were identical across arms.

We ran every design except the store-age and cross-vendor tests on October 9, 2026. The technique-library test made 1,602 model calls, using 79.6 million input tokens and 6.6 million output tokens for its 720 episodes.

A.3 Prompts

Every reader in every design receives the same system prompt, verbatim:

```
You are an expert GPU kernel engineer. You are given a PyTorch reference module `Model` from KernelBench.
Write `ModelNew`: a drop-in replacement (same __init__ arguments, same forward inputs and outputs) that is faster
because it uses your own GPU kernels.

Environment (fixed):
- One NVIDIA A100-SXM4-40GB (sm_80, 108 SMs, 40 MB L2, ~1.5 TB/s HBM). CUDA 12.9, PyTorch 2.9.1, Triton 3.5.1,
  Python 3.10.
- Your kernels: CUDA C++ through torch.utils.cpp_extension.load_inline, and/or Triton (@triton.jit). At least one
  of your own kernels must actually run in every forward call. You may still call torch ops (cuBLAS/cuDNN through
  torch) for the parts you do not replace.
- Numerics are strict float32: outputs must match the reference within atol=rtol=1e-4 on fresh random inputs and
  fresh random weights (initialised like the reference: build parameters in the same order with the same nn
  layers, e.g. self.conv = nn.Conv2d(...), so the evaluator's seeded init gives identical weights). TF32, fp16 and
  bf16 shortcuts usually fail this tolerance; cuBLAS/cuDNN TF32 is disabled and global flags cannot be changed.
- Score: speedup = reference eager time / your time, measured by a protected evaluator (cold L2, fresh inputs on
  every call, device-synchronised timing interleaved with the reference). Correct and faster than eager with 95%
```

```
  confidence counts as a win; the size of the speedup is your score.

Rules (enforced; a violation rejects the submission):
- Imports only from: torch, torch.nn, torch.nn.functional, torch.utils.cpp_extension (load_inline only), triton,
  triton.language, math, numpy, typing, functools, itertools, operator, collections, dataclasses, enum.
- No try/except, no global statements, no streams, events, CUDA graphs, torch.compile, torch.jit, torch.backends,
  profiler, file/OS/process access, exec/eval/open/setattr/getattr-with-variables, no private (_x) or dunder
  attributes except __init__, no assignment to attributes other than self.<x>, no class named Model.
- Do not cache results across calls, do not modify the input tensors in place, do not return a tensor subclass.
- Native code: no Python C API or pybind11 types, no stream/graph/event APIs, no host system calls or files.
  Launch kernels on the current stream (at::cuda::getCurrentCUDAStream()) or the default stream.
- load_inline compile time: about 35 s when cuda_sources include ATen/torch headers, about 5 s when cuda_sources
  are plain CUDA (#include <cuda_runtime.h>) with launcher functions that take raw pointers and a cudaStream_t,
  while the torch::Tensor wrapper lives in cpp_sources (declare the launcher there). Either works.

Reply format: think as needed, then give exactly ONE ```python code block containing the complete module
(imports, kernels, class ModelNew). End with a final line that is either
ACTION: CHECK    (run the local check: compile, rules and correctness on one public input; no timing)
or
ACTION: SUBMIT   (final: the protected evaluator scores this code once).
```

Local checks return under the line “Local check result for your last code (correctness only, public input, no timing):”. The served store follows the reference module under the headers of Figure 3; late delivery appends it to the first local check under the header quoted in Section 4.8.

The zero-shot self-gate of Section 4.6 asks Opus 5.5, once per problem and reader, with the problem’s reference module and its rendered store in the slots shown:

```
An AI model writes ModelNew, a faster drop-in replacement for a KernelBench PyTorch reference Model, using its
own CUDA (load_inline) or Triton kernels. A protected evaluator checks correctness (strict fp32, 1e-4) and times
it against PyTorch eager; the score is the speedup if correct (capped at 10), else 0. A memory store carried
from earlier attempts on OTHER problems (written from an earlier model's verified work: environment lessons,
family pitfalls, recipes with measured speedups, related verified kernels) can be shown to the model before it
starts, whole, or withheld. A store can help one model while hurting another, and help on one problem while
hurting on another.

Model: <reader name>
Problem (<family, level, op count, source lines, store item kinds, max recipe speedup>):
<reference module>

The store for this problem:
<rendered store>

Decide whether to SHOW this store to THIS model for THIS problem (to maximise its expected score) or WITHHOLD
it. End your answer with exactly one line: DECISION: SHOW or DECISION: WITHHOLD
```

The reader-level rule of Section 4.8 uses the same prompt with a digest of the randomized outcomes inserted after the first paragraph.

A.4 Store excerpt

The rendered headers vary by problem family and related kernel. For a matrix-multiplication problem they read:

```
## Kernel memory (items verified by the protected evaluator on training problems; may
not all apply)
### Lessons about this environment (from official checker verdicts and verified
outcomes)
### Pitfalls seen on earlier matmul problems (wrong results or slower than eager)
### Verified recipes for matmul problems (speedups measured by the protected evaluator)
### A verified kernel for a related matmul problem
    (L1_015 Matmul_for_lower_triangular_matrices, 4.58x):  adapt what applies
```

A verbatim checker lesson reads: “Checker rule: getattr needs a literal, non-private attribute name.” A lesson written by Opus 5.5 from Opus 4.8’s failures, the exact-fp32 rule of Section 4.5: “Pitfall: Output mismatch beyond 1e-4 occurred repeatedly. Fix by forcing exact fp32 tl.dot(allow_tf32=False) and exact masking: K bounds at block granularity, tril mask, and zero-initialized output for skipped tiles.” The fixed process rule: “Process: never SUBMIT code that has not passed ACTION: CHECK in this episode. . . . ” A family pitfall written by Opus 5.5: “Pitfall: Avoid launching a per-forward weight-transform kernel (flip/transpose) before a library call. Cache the transformed weight once, or fuse the transform, since extra launches and memory traffic make it slower than eager.” One recipe, written by Opus 5.5 from an Opus 4.8 solution, reads:

[1.68x on ConvTranspose3d_BatchNorm_AvgPool_AvgPool] Keep the ConvTranspose in cuDNN via torch. Fold training-mode BatchNorm into a per-channel affine: one Triton program per (n,c) row loops over contiguous spatial data in 1024-wide blocks, accumulating fp32 sum and sum-of-squares. Finish mean/var/scale/shift with tiny torch ops. Because BN is affine, apply it after pooling: replace the two 2x pools with one fused 4x4x4 average kernel. Use one thread per output, int64 index math, static_range unrolled loads, and scale*avg+shift at store.

Related kernels appear as verified code under a header such as "A verified kernel for a related conv_transpose problem (L2_078 ConvTranspose3d_Max_Max_Sum, 1.00x): adapt what applies," beginning `import torch` … `import triton` … `@triton.jit def fused_kernel(x_ptr, out_ptr, N, C, D, H, W, ...)`.

B Additional results

B.1 Per-problem head-to-head means

Table 12: **The whole store lowers Opus 5.5 on nine of ten head-to-head problems and raises it on none.** Mean capped speedup by problem, 4 repetitions, for Opus 5.5 (top) and Opus 4.8 (bottom).

Problem	bare	show-all	provenance	selector	digest	tables	annotated
Claude Opus 5.5							
L2_014	1.00	1.00	1.00	1.00	1.01	1.00	1.01
L2_022	1.34	1.03	1.04	1.03	1.16	2.65	1.04
L2_030	1.27	1.03	1.02	1.03	1.24	1.03	1.03
L2_033	1.13	1.01	1.01	1.01	1.01	1.01	1.01
L2_050	10.00	10.00	10.00	10.00	10.00	10.00	5.82
L2_061	1.04	0.94	1.03	1.03	0.94	1.03	0.94
L3_010	0.90	0.76	0.85	0.77	0.74	0.77	0.71
L3_023	0.81	0.79	0.59	0.74	0.73	0.90	1.02
L3_032	7.22	6.82	5.24	6.49	5.89	5.31	6.75
L3_040	8.38	6.84	6.47	7.33	6.78	5.90	7.68
Claude Opus 4.8							
L2_014	0.00	0.75	0.75	1.00	0.75	0.75	0.75
L2_022	1.03	1.04	1.03	1.03	1.03	1.03	1.03
L2_030	0.51	1.03	1.02	1.03	1.03	1.02	1.02
L2_033	0.99	0.99	1.00	0.99	1.00	0.99	0.99
L2_050	0.83	1.09	1.10	1.10	1.10	1.10	1.10
L2_061	0.99	1.02	1.02	1.02	1.02	1.02	1.02
L3_010	0.38	0.80	0.81	0.73	0.72	0.87	0.82
L3_023	0.20	0.29	0.30	0.29	0.29	0.30	0.30
L3_032	0.88	0.81	0.64	0.86	0.85	0.88	0.89
L3_040	0.12	0.28	0.12	0.11	0.78	0.79	0.71

B.2 Behavioral readouts

Table 13: **Shown the store, Opus 5.5 writes less CUDA and fewer tokens and submits sooner.** Problem-paired differences. "Both" uses 1,256 Opus 5.5 episodes over 120 problems (578 without, 678 with the store); "Randomized" uses 920 episodes over 92 problems.

Readout	Set	No store	Store	Difference
Correct episodes	Both	576/578	678/678	$+0.003$
Log speedup given correct	Both			-0.126 $[-0.185, -0.075]$
Output tokens per episode	Both	9,597	7,142	$-2{,}588$ $[-3{,}286, -1{,}940]$
Submit right after first check	Both	76%	86%	$+10$ pts
Episodes in CUDA C++	Randomized	202/466	60/454	$43.4\% \to 13.2\%$
Token kernels ($< 5\%$ own runtime)	Randomized	92/466	145/454	$19.7\% \to 31.9\%$
Store coefficient, problem fixed effects	Randomized			-0.106 $[-0.168, -0.053]$
+ CUDA indicator	Randomized			-0.053 $[-0.099, -0.013]$
+ CUDA and token-kernel indicators	Randomized			-0.024 $[-0.065, +0.015]$

B.3 The two L2_074 kernels

The successor's bare kernel (episode 609, $9.63\times$) is a single CUDA C++ kernel compiled with `load_inline`. Weights for four output channels are staged in shared memory as `float4`; each thread accumulates, for two pooled

positions, the eight parity outputs of the pooling window from a $2 \times 2 \times 3$ input neighborhood, then applies the epilogue and the maximum in registers. Its `forward` falls back to the reference operators for other shapes; on the benchmark's shapes the kernel runs. The listing is verbatim with indentation halved.

```
__global__ void fused_kernel(const float* __restrict__ x, const float* __restrict__ w,
  const float* __restrict__ bias, const float* __restrict__ mul, float* __restrict__ out,
  int D, int H, int W)
{
  __shared__ float4 ws[16*27];
  const int ocg = blockIdx.y;
  const int n = blockIdx.z;
  const int tid = threadIdx.y * 16 + threadIdx.x;
  for (int i = tid; i < 16*27; i += 256) {
    int ci = i / 27, tap = i % 27;
    float4 v;
    v.x = w[((ci*32) + ocg*4 + 0)*27 + tap];
    v.y = w[((ci*32) + ocg*4 + 1)*27 + tap];
    v.z = w[((ci*32) + ocg*4 + 2)*27 + tap];
    v.w = w[((ci*32) + ocg*4 + 3)*27 + tap];
    ws[i] = v;
  }
  __syncthreads();
  const int htiles = H / 16;
  const int pd = blockIdx.x / htiles;
  const int ph = (blockIdx.x % htiles) * 16 + threadIdx.y;
  const int w0 = 2 * threadIdx.x;
  const bool vd = (pd + 1) < D;
  const bool vh = (ph + 1) < H;
  const bool vw = (w0 + 2) < W;

  float acc[2][8][4];
  #pragma unroll
  for (int j = 0; j < 2; j++)
    #pragma unroll
    for (int p = 0; p < 8; p++)
      #pragma unroll
      for (int o = 0; o < 4; o++) acc[j][p][o] = 0.f;

  const long long DHW = (long long)D * H * W;
  const float* xb = x + (long long)n * 16 * DHW;
  #pragma unroll 1
  for (int ci = 0; ci < 16; ci++) {
    const float* xc = xb + ci * DHW;
    float xin[2][2][3];
    #pragma unroll
    for (int dd = 0; dd < 2; dd++)
      #pragma unroll
      for (int hh = 0; hh < 2; hh++)
        #pragma unroll
        for (int ww = 0; ww < 3; ww++) {
          bool valid = (dd == 0 || vd) && (hh == 0 || vh) && (ww < 2 || vw);
          xin[dd][hh][ww] = valid ? __ldg(xc + ((long long)(pd + dd) * H + ph + hh) * W + w0 + ww) : 0.f;
        }
    const float4* wc = ws + ci * 27;
    #pragma unroll
    for (int kd = 0; kd < 3; kd++) {
      const int a = (kd == 1) ? 0 : 1;
      const int id = (kd == 0) ? 1 : 0;
      #pragma unroll
      for (int kh = 0; kh < 3; kh++) {
        const int b = (kh == 1) ? 0 : 1;
        const int ih = (kh == 0) ? 1 : 0;
        #pragma unroll
        for (int kw = 0; kw < 3; kw++) {
          const int c = (kw == 1) ? 0 : 1;
          const int iw = (kw == 0) ? 1 : 0;
          const int par = a * 4 + b * 2 + c;
          float4 wv = wc[(kd * 3 + kh) * 3 + kw];
          #pragma unroll
          for (int j = 0; j < 2; j++) {
            float xv = xin[id][ih][iw + j];
            acc[j][par][0] = fmaf(wv.x, xv, acc[j][par][0]);
            acc[j][par][1] = fmaf(wv.y, xv, acc[j][par][1]);
            acc[j][par][2] = fmaf(wv.z, xv, acc[j][par][2]);
            acc[j][par][3] = fmaf(wv.w, xv, acc[j][par][3]);
          }
        }
      }
    }
  }
  const int PHW = H * W;
  #pragma unroll
  for (int o = 0; o < 4; o++) {
    const int oc = ocg * 4 + o;
    const float bv = bias[oc];
    const float mv = mul[oc];
    #pragma unroll
    for (int j = 0; j < 2; j++) {
      float mx = -3.402823466e38f;
```

```
      #pragma unroll
      for (int p = 0; p < 8; p++) {
        float v = acc[j][p][o] + bv;
        v = v > 0.f ? v : 0.2f * v;
        v = v * mv;
        v = v > 0.f ? v : 0.2f * v;
        mx = fmaxf(mx, v);
      }
      out[((long long)(n * 32 + oc) * D + pd) * PHW + ph * W + w0 + j] = mx;
    }
  }
}
```

The successor's kernel with the store (episode 814, 1.27×) keeps `nn.ConvTranspose3d` and adds one Triton kernel for the epilogue and pool. The listing is verbatim with indentation halved and the imports omitted.

```
@triton.jit
def fused_pool_kernel(x_ptr, m_ptr, out_ptr, C, D, H, W, OD, OH, OW, total,
           BLOCK: tl.constexpr):
  pid = tl.program_id(0).to(tl.int64)
  idx = pid * BLOCK + tl.arange(0, BLOCK).to(tl.int64)
  mask = idx < total

  ow = idx % OW
  t = idx // OW
  oh = t % OH
  t = t // OH
  od = t % OD
  nc = t // OD
  c = nc % C

  mult = tl.load(m_ptr + c, mask=mask, other=1.0)
  mult2 = mult[:, None]

  j = tl.arange(0, 2).to(tl.int64)
  D64 = D.to(tl.int64)
  H64 = H.to(tl.int64)
  W64 = W.to(tl.int64)
  mask2 = mask[:, None] & (j[None, :] < 2)

  res = tl.full((BLOCK,), float("-inf"), tl.float32)
  for dz in tl.static_range(2):
    iz = od * 2 + dz
    for dy in tl.static_range(2):
      iy = oh * 2 + dy
      rowbase = ((nc * D64 + iz) * H64 + iy) * W64 + ow * 2
      off = rowbase[:, None] + j[None, :]
      v = tl.load(x_ptr + off, mask=mask2, other=0.0)
      v = tl.where(v > 0, v, v * 0.2)
      v = v * mult2
      v = tl.where(v > 0, v, v * 0.2)
      res = tl.maximum(res, tl.max(v, axis=1))
  tl.store(out_ptr + idx, res, mask=mask)


class ModelNew(nn.Module):
  def __init__(self, in_channels, out_channels, kernel_size, stride, padding, output_padding, multiplier_shape):
    super().__init__()
    self.conv_transpose = nn.ConvTranspose3d(in_channels, out_channels, kernel_size, stride=stride,
                      padding=padding, output_padding=output_padding)
    self.multiplier = nn.Parameter(torch.randn(multiplier_shape))

  def forward(self, x):
    y = self.conv_transpose(x).contiguous()
    N, C, D, H, W = y.shape
    OD, OH, OW = D // 2, H // 2, W // 2
    out = torch.empty((N, C, OD, OH, OW), device=y.device, dtype=y.dtype)
    total = N * C * OD * OH * OW
    m = self.multiplier.reshape(-1).contiguous()
    BLOCK = 256
    grid = (triton.cdiv(total, BLOCK),)
    fused_pool_kernel[grid](y, m, out, C, D, H, W, OD, OH, OW, total, BLOCK=BLOCK, num_warps=4)
    return out
```