KERNELSHIFT: BENCHMARKING KERNEL OPTIMIZATION ACROSS EVOLVING HARDWARE

Anonymous authors
Paper under double-blind review

ABSTRACT

Efficient AI computation depends on hardware that *evolves over time*, shifting the strategies needed to implement performant code. For CPU kernels, a key evolving factor is the set of available *architectural extensions*, which expose specialized hardware instructions that change the techniques used to optimize kernels. We study how effectively coding agents use these new, potentially unfamiliar extensions to write fast code. We introduce **KERNELSHIFT**, a benchmark for evaluating AI coding agents on kernel optimization across evolving CPU architectures. We design an agent-assisted, human-verified pipeline for transforming code repositories into benchmark instances, together with infrastructure that lets agents iteratively optimize code using feedback from the target hardware. Using this pipeline, we construct 85 kernels from real-world, actively developed repositories (ncnn, llama.cpp, Arm SIMD Loops, and KleidiAI), including inference workloads derived from real model executions. KERNELSHIFT supports incorporating new kernels and hardware configurations as libraries and processors evolve. We evaluate agents on proprietary and open-weight models across progressively newer architectural extensions (Arm Neon, SVE, SVE2, and SME2), examining how performance changes as hardware evolves. On established ISAs, agents exploit residual optimization headroom, with the strongest system achieving a $2.74\times$ geometric-mean speedup over the ncnn production reference on SVE. In end-to-end inference of two LLMs, agent-written kernels achieve a $1.3\times$ prefill speedup over llama.cpp's fast path. However, current agents do not yet reliably adapt to new extensions that require unfamiliar programming models, such as SME2's matrix tiles.

1 INTRODUCTION

Modern ML workloads run across an increasingly diverse range of hardware, from CPUs to GPUs with different architectures and capabilities (Reddi et al., 2020). As these platforms continue to evolve, the optimization strategies needed to use them efficiently evolve as well. Meanwhile, AI coding agents are increasingly being used to optimize performance-critical code. For example, frontier labs such as OpenAI report using AI models to autonomously rewrite and optimize production GPU kernels (Ferrari et al., 2026; Novikov et al., 2025). As AI agents take on more performance-critical optimization, an important question is whether they can adapt their optimization strategies as the underlying hardware *evolves over time*.

In CPUs, hardware evolution often takes the form of new *architectural extensions*, which introduce new instructions and expose them to programmers through a new interface. Extensions can improve performance, but exploiting them often requires fundamentally restructuring a kernel's implementation. For example, Arm's matrix extensions support accumulating products directly into 2D matrix tiles, enabling implementations of matrix multiplication that differ from those based on earlier 1D vector instructions (Arm, 2026; Remke & Breuer, 2024). An agent can therefore produce correct but suboptimal code if it relies on older vector instructions when newer matrix instructions would be advantageous.

To study kernel optimization under evolving hardware, we introduce KERNELSHIFT (Figure 1 (left)), a benchmark for evaluating AI coding agents across evolving CPU architectures. The key idea is to evaluate how agents optimize matched kernels, where coverage permits, across

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

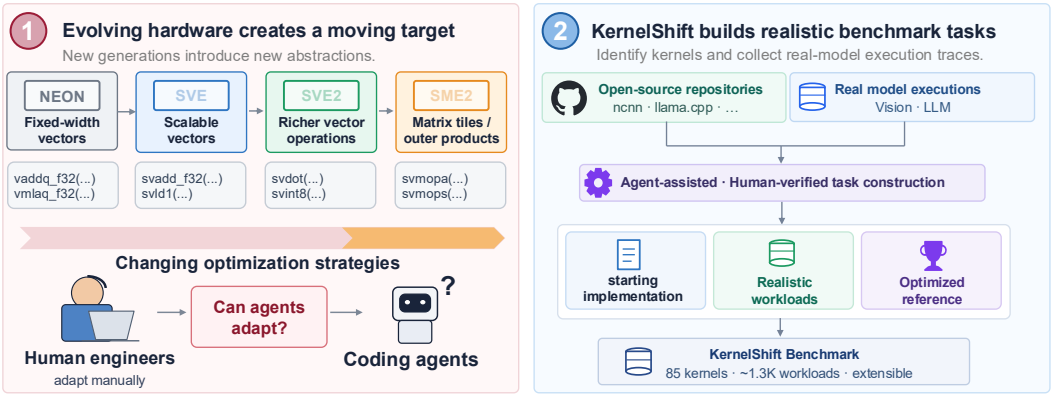


Figure 1: **Overview of KERNELSHIFT. (Left)** Evolving hardware changes the space of performant kernel implementations, requiring both human engineers and coding agents to continually adapt their strategies. **(Right)** To study this adaptation, KERNELSHIFT constructs benchmark tasks from real-world sources and model executions using an agent-assisted, human-verified pipeline.

successive *architectural extensions* that shift the space of performant implementations. Specifically, we evaluate agents across Arm's vector and matrix extensions (i.e., Neon, SVE, SVE2, and SME2), which provide instructions for parallelizing computations common in machine learning workloads (Stephens et al., 2017; Arm, 2026). Our work builds on a growing literature on hardware-aware kernel optimization (He et al., 2025; Wen et al., 2025; Zang et al., 2026), with a specific focus on sequences of hardware changes that expose progressively newer programming interfaces.

We design an agent-assisted, human-verified pipeline with source-specific adapters for transforming code repositories into benchmark instances (Figure 1 (right)), together with infrastructure that lets agents iteratively optimize code using compilation and performance feedback from the target hardware (Figure 2). Using this pipeline, we construct 85 kernel specifications from real-world, actively developed repositories, including workloads derived from real model executions. The construction pipeline and evaluation infrastructure are designed to incorporate new kernels and hardware configurations as libraries and processors evolve.

We evaluate coding agents on proprietary and open-weight models across progressively newer Arm architectural extensions. We measure both optimization performance and their use of newly available instructions, comparing their implementations against compiler-optimized code, production-library implementations, and expert kernels where available. Agents readily find headroom on established vector extensions: on SVE, the strongest system achieves a $2.74\times$ geometric-mean speedup over the ncnn production reference within 40 tool calls. But this ability does not reliably transfer to newer programming models: on SME2, models that fail to adopt matrix-tile accumulation fall far behind expert implementations.

Our main contributions are:

- **KERNELSHIFT**, a benchmark of real-world CPU kernels for evaluating coding agents across evolving hardware and programming interfaces.
- **An agent-assisted, human-verified construction pipeline and evaluation framework** that supports adding kernels and hardware configurations as software libraries and processors evolve.
- **An empirical study of coding agents across Arm architecture extensions**, measuring both optimization performance and agents' ability to exploit newly available hardware instructions.

2 KERNELSHIFT

Our goal is to evaluate agents' ability to write performant code for *evolving hardware*. We introduce our problem setting, motivate our use of *architectural extensions* to distinguish stages of CPU hardware evolution, and describe our automated pipeline for collecting benchmark instances.

Hardware-aware kernel optimization. We consider the setting of *hardware-aware kernel optimization*. In this setting, a *kernel* is a program that performs a specific computation, such as a matrix multiplication. Each task consists of an initial kernel implementation x, together with context c. The context includes the function signature, symbolic tensor shapes, and data types; concrete workload dimensions and input values remain hidden (Appendix C.3).

The kernel is executed on target hardware, $h = (I, m)$. Here, I denotes the target *instruction-set architecture* (ISA), including base instructions and those introduced by supported *architectural extensions*. Programmers can access specialized instructions through *intrinsics*: compiler-supported operations exposed in C/C++. The *microarchitecture* m is a concrete processor design that implements I on physical hardware. In general, the goal is to develop an agent A that takes in an initial implementation, context, and hardware, and produces an optimized implementation $x' = A(x, c, h)$:

$$\max_{A} \mathbb{E}_{(x,c)\sim\mathcal{D},\, h\sim\mathcal{H}} \left[s\big(x, A(x, c, h)\big) \right], \tag{1}$$

where $\mathcal{D}$ is a distribution of optimization tasks and $\mathcal{H}$ is a distribution of hardware configurations.

For a fixed task and target hardware, the score $s(x, x')$ is the geometric mean speedup over the task's optimized reference across n hidden workloads, which specify input configurations such as matrix dimensions. That is, let t_i^{ref} and t_i' denote the runtimes of the optimized reference and the agent implementation on workload i. The score is:

$$s = \left(\prod_{i=1}^{n} \frac{t_i^{\text{ref}}}{t_i'} \right)^{1/n}, \tag{2}$$

with $s = 0$ if the agent implementation fails any correctness check (Appendix B.3).

Evolving hardware. Rather than sampling hardware configurations to approximate Equation 1, we are interested in sequences of hardware configurations that introduce new functionality to a common base architecture,

$$h_1 \longrightarrow h_2 \longrightarrow \cdots \longrightarrow h_T. \tag{3}$$

The kernel's desired functionality remains fixed across the sequence, while the implementations that realize the functionality efficiently may change substantially.

Specifically, we primarily focus on changes to the available instruction set I through new extensions, since new extensions can introduce different ways to optimize computations and hence cause large shifts in the space of performant programs. For example, as illustrated in Figure 1, Arm's *Neon* extension exposes fixed-width vector operations, while the later *scalable vector extensions* (SVE, SVE2) support code that adapts to different vector widths and provide additional vector operations. The more recent *scalable matrix extension* (SME) introduces dedicated matrix storage and instructions for accumulating vector outer products into matrix tiles, enabling new ways to implement matrix multiplication.

Motivated by these architectural shifts, we evaluate agents' kernel optimization performance across such evolving extensions. In particular, we investigate whether choice of extension impacts agent performance, and whether agents can effectively leverage newer intrinsics that are underrepresented in their training data. Therefore, we construct an automated benchmark framework that seamlessly accommodates new extensions as they arise. The benchmark design and automated generation techniques are detailed next.

2.1 Agent-Assisted Task Construction

Our construction framework turns performance-critical kernels from open-source projects into benchmark tasks. For each kernel, it constructs a portable C++ starting implementation x, the context c exposed to the agent, hidden workloads, and, when available, an optimized reference. Tasks are constructed by coding agents following fixed procedures backed by deterministic generation and validation scripts. Human input is limited to a few decisions: selecting kernels, approving workloads where no model execution can be traced, and reviewing the resulting expert references. Appendix B documents the complete construction process.

Table 1: **Composition of KERNELSHIFT.** We apply our kernel collection pipeline on common or vendor-provided repositories, resulting in a wide range of kernels.

Source	Op types	Precisions	Expert reference	#Kernels
ncnn	conv, dwconv, pool	FP32, INT8 (w8a8ch)	ncnn Neon	23
llama.cpp	gemm, moe, attn, rmsnorm	FP32, BF16, q8_0, q4_K_M	ggml	26
SIMD Loops	loops (reduction, sort, string, ...)	fp/int/uint	Arm SVE	26
KleidiAI	gemm, conv, dwconv (NHWC)	FP32, FP16, BF16, INT4	KleidiAI	10 (3 SVE, 7 SME2)
Total				85

Kernel and reference construction. For each source repository, a construction agent follows a fixed procedure (Appendix B.1) to identify candidate kernel computations and extract their fixed configuration parameters and variable workload dimensions. It then constructs a portable C++ starting implementation using simple loop-based code. When the repository contains an optimized implementation of the same kernel, the agent also extracts it as the optimized reference.

Workload construction. To capture realistic workloads, we derive input configurations from real model executions whenever possible. For model-derived kernels, the construction agent first identifies the source model in which the kernel appears. Fixed configuration parameters then come from this model, while dimensions are collected from execution traces. For example, ncnn workloads come from vision-model executions, while llama.cpp workloads are collected during LLM inference on ShareGPT prompts (ShareGPT, 2023). We keep the observed tensor shapes rather than the original activation values, under the assumption that kernel performance is driven primarily by tensor dimensions and operator configuration rather than the specific numerical values of the activations. We then deduplicate and subsample the collected shapes to form the hidden evaluation workloads. For kernels without a clearly identifiable source model, we instead synthesize workloads in two groups: correctness-oriented cases that cover corner cases, and performance-oriented cases that probe efficiency over representative problem sizes. More details appear in Appendix B.2.

Finalizing benchmark tasks. Finally, we package the starting implementation, agent-visible context, evaluation workloads, and the optimized reference from the source library, when available, into a self-contained benchmark task. Each task is validated by compiling and running the starting implementation on the target hardware before it is included in KERNELSHIFT.

2.2 BENCHMARK COMPOSITION AND COVERAGE

We apply the construction framework from Section 2.1 to several widely used, performance-critical open-source repositories.

Composition. KERNELSHIFT currently contains kernel tasks derived from four sources (Table 1): ncnn (Tencent, 2017), llama.cpp (Gerganov & ggml-org contributors, 2023), Arm SIMD Loops (Arm, 2025), and KleidiAI (Arm, 2024). Together, they cover computer-vision and LLM inference kernels, general SIMD workloads, and vendor-optimized microkernels across a range of data types and optimization patterns. The ncnn and llama.cpp tasks are derived from 11 source models, providing kernel configurations representative of real model architectures. Appendices A and A.2 provide the full composition and per-generation coverage; Appendix A.3 lists every kernel.

Workloads. KERNELSHIFT contains approximately 1.3K workloads across the 85 kernel specifications, constructed as described in Section 2.1 and spanning a broad range of tensor dimensions and sequence lengths.

Hardware and reference coverage. The 85 kernels form the union across targets: every task can be instantiated on each target, but an optimized reference is not available for every source on every generation, so each generation uses the tasks that have one. For each task, the reference is a strong production or expert implementation available on the target (Appendix A.2).

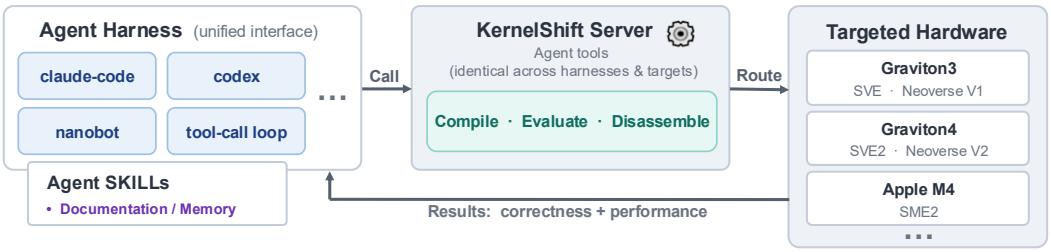


Figure 2: **Evaluation architecture of KERNELSHIFT.** A unified server exposes the same tools and skills to all agent harnesses and routes their requests to evaluators running on the target hardware.

2.3 EVALUATION ENVIRONMENT

KERNELSHIFT uses a centralized control plane to run agent harnesses and coordinate evaluation across distributed target machines (Figure 2).

Hardware-in-the-loop evaluation. Each target platform is backed by an evaluator running directly on its hardware. When an agent calls a tool, the request is routed to this evaluator, which compiles and executes the candidate on the target machine and returns the feedback, so that correctness and performance measurements reflect the hardware being evaluated. This design decouples the agent harness from target-specific execution, allowing the same agent setup to be used across hardware platforms. Appendix C gives the deployment configurations.

Unified agent interface. To compare different agent systems under the same environment, KERNELSHIFT exposes a harness-agnostic interface through which all agents access the benchmark. The interface provides the same three benchmark tools to every harness: *compile*, *evaluate*, and *disassemble*, plus a utility that returns the best result found so far. These tools allow agents to iteratively refine their solutions with correctness, performance, and low-level instruction feedback from the target hardware. Appendix C.3 provides the complete prompts, skills, and tool schemas.

Measurement and integrity. Correctness is checked before performance is measured, and both agent implementations and comparison baselines are tested on the same target machine under the same timing protocol. We use wall-clock execution time as the common performance measure across platforms, since hardware cycle counters are not consistently available on all target systems. Performance is reported as the speedup of a kernel over its expert reference on the same machine. Whenever a new target instance is provisioned, we re-run the baselines on that instance to reduce performance differences across machines. Measurements use repeated runs on a single core, and each evaluation runs in an isolated process to prevent interference between evaluations. We also reject implementations that violate the benchmark constraints, such as using multithreading. Full timing, isolation, and integrity details are provided in Appendix C.4.

3 EXPERIMENTAL SETUP

Models and harnesses. We evaluate six frontier language models: Claude Opus 4.8 (Anthropic, 2026b) and Claude Sonnet 4.6 (Anthropic, 2026c), GPT-5.6-sol and GPT-5.6-luna (Ferrari et al., 2026), DeepSeek-V4-Pro (DeepSeek-AI, 2026), and GLM-5.3 (Zhipu AI, 2026). We pair these models with four agent harnesses: Claude Code (Anthropic, 2025), Codex (OpenAI, 2025), nanobot (HKUDS, 2025), and a minimal tool-call loop. We evaluate Claude Code and Codex with their own models, while nanobot provides a common harness across all six models to disentangle model and harness effects. The end-to-end study (Section 4.5) additionally uses Claude Fable 5.1 (Anthropic, 2026a) with Claude Code. Appendix C.3 contains full model and harness configurations.

Task sets. Our experiments draw from the 85-kernel benchmark, with generation-specific coverage: the SVE comparison contains 78 kernels (23 ncnn, 26 llama.cpp, 26 SIMD Loops, and 3 KleidiAI). We use a 32-kernel subset for ablations and controlled studies.

Table 2: **Results on SVE (Graviton3)**. Each cell reports the geometric mean over kernels of Speedup@40 against the source's expert reference and, in parentheses, the fraction of kernels that beat that reference (>1). The best speedup in each column is in bold.

Harness	Model	Geomean Speedup@40 (beat-ref %) vs. expert reference			
		ncnn (23)	llama.cpp (26)	SIMD Loops (26)	KleidiAI (3)
Claude Code	Claude Opus 4.8	2.574 (96%)	1.779 (73%)	1.807 (96%)	**0.887** (67%)
Claude Code	Claude Sonnet 4.6	1.934 (83%)	1.243 (46%)	1.731 (100%)	0.750 (67%)
Codex	GPT-5.6-sol	1.815 (78%)	1.297 (62%)	1.760 (96%)	0.646 (33%)
Codex	GPT-5.6-luna	0.674 (35%)	0.627 (27%)	1.255 (77%)	0.312 (0%)
nanobot	Claude Opus 4.8	2.597 (91%)	1.708 (73%)	1.685 (96%)	0.851 (33%)
nanobot	Claude Sonnet 4.6	2.046 (87%)	1.449 (62%)	1.612 (92%)	0.464 (0%)
nanobot	GPT-5.6-sol	2.156 (87%)	1.334 (58%)	**1.891** (100%)	0.638 (0%)
nanobot	GPT-5.6-luna	1.110 (57%)	0.878 (31%)	1.305 (77%)	0.372 (0%)
nanobot	DeepSeek-V4-Pro	1.091 (57%)	0.850 (46%)	1.405 (85%)	0.710 (33%)
nanobot	GLM-5.3	**2.743** (91%)	**1.954** (85%)	1.586 (96%)	0.778 (33%)
tool-call loop	Claude Opus 4.8	2.308 (83%)	1.828 (73%)	1.590 (88%)	0.797(67%)
tool-call loop	Claude Sonnet 4.6	2.043 (87%)	1.915 (81%)	1.734 (100%)	0.660(33%)
tool-call loop	GPT-5.6-sol	1.999 (91%)	1.671 (81%)	1.693 (96%)	0.678 (67%)
tool-call loop	GPT-5.6-luna	0.955 (48%)	0.854 (31%)	1.407 (77%)	0.505 (33%)
single-shot, no tools (Claude Opus 4.8)[†]		0.577 (35%)	0.324 (8%)	0.849 (42%)	0.517 (0%)
[†]For correctness failures, we substitute the starter-kernel score.					
Reference tiers					
scalar + `-O2` (no vectorization)		0.190 (0%)	0.090 (8%)	0.273 (4%)	0.054 (0%)
scalar + `-O2` (Neon auto-vectorized)		0.191 (0%)	0.117 (12%)	0.424 (19%)	0.083 (0%)
scalar + `-O3`, SVE enabled		0.229 (9%)	0.126 (15%)	0.492 (27%)	0.098 (0%)

Hardware and budgets. We evaluate SVE on Graviton3 (AWS c7g), SVE2 on Graviton4 (AWS c8g), and SME2 on Apple M4. Each run receives a budget of $T = 40$ tool calls, with *compile*, *evaluate*, and *disassemble* calls all counting toward the budget. For test-time scaling experiments, we extend the budget to 100 tool calls.

Metrics. Our primary metric, Speedup@40, is the score s (Section 2) of the best correct implementation within 40 calls, averaged geometrically over kernels. We also report the fraction of kernels that beat the reference, compiler baselines, and performance versus budget.

4 Results

4.1 Agents find substantial speedups on fixed workload sets

On fixed sets of workloads, strong coding agents can exploit substantial optimization headroom in production kernels. On SVE, the best systems reach 2.74× the ncnn reference, 1.95× the llama.cpp reference, and 1.89× the SIMD Loops reference (Table 2). In contrast, compiler-based automatic optimization remains far behind expert references, with even `-O3` and SVE reaching at most 0.49×.

Highly tuned vendor kernels remain substantially more competitive. On the three KleidiAI kernels with SVE expert references, no evaluated system matches the vendor implementations in geometric mean; the best, Claude Code with Claude Opus 4.8, reaches 0.89×, beating the reference on two kernels but falling behind on the third. Agents exploit headroom in general-purpose production libraries, but matching specialized vendor kernels is harder.

4.2 Adapting to Evolving Hardware

Agents struggle more with the programming-model shift to SME2. The biggest drop appears on SME2, which introduces streaming mode and ZA matrix tiles instead of just adding more vector instructions. On the KleidiAI tasks, performance relative to the expert reference falls noticeably from SVE to SME2: the SME2-to-SVE ratios are 0.80, 0.49, 0.23, and 0.19 for Claude Opus 4.8, Claude Sonnet 4.6, DeepSeek-V4-Pro, and GLM-5.3, respectively (Table 4). The fact that all four

Table 3: **Instruction-set ladders on fixed hardware.** Holding the machine and task set fixed, we progressively expose newer instructions; every arm is scored against the same reference. Per-arm compile flags are in Appendix C.2.

Arm	Instructions available	Speedup@40
Graviton4, 32-kernel subset		
L1	Neon only	1.088
L2	+ SVE	0.957
L3	+ SVE2	1.101
Apple M4, KleidiAI SME2 set		
L4	Neon only	0.205
L5	+ streaming SVE (no ZA)	0.090
L6	+ ZA / `FMOPA`	0.086

Table 4: **Cross-generation adaptation against expert references.** Speedup@40 (nanobot) against the expert implementation of the older (Before) and newer (After) generation; Ratio = After / Before.

Model	Before	After	Ratio
KleidiAI: SVE (3 kernels) → SME2 (7)			
Claude Opus 4.8	0.851	0.683	0.80
Claude Sonnet 4.6	0.464	0.226	0.49
DeepSeek-V4-Pro	0.710	0.161	0.23
GLM-5.3	0.778	0.149	0.19
SIMD Loops: SVE (7) → SVE2 (7)			
Claude Opus 4.8	1.817	1.942	1.07
Claude Sonnet 4.6	1.707	1.718	1.01
DeepSeek-V4-Pro	1.787	1.387	0.78
GLM-5.3	1.806	1.974	1.09

models drop suggests that agents have a harder time keeping up when a new extension requires a different programming style, rather than simply exposing more instructions. The fixed-machine ladder on Apple M4 shows the same effect with the machine and reference held fixed: exposing streaming SVE and then full SME2 lowers Speedup@40 from 0.205 (Neon only) to 0.090 and 0.086 (Table 3). Agents keep writing streaming and ZA code incorrectly, so giving them more intrinsics that they cannot yet use correctly lowers performance. This evidence covers the seven KleidiAI kernels with an SME2 expert reference.

A closer look at one SME2 kernel shows what this gap looks like in practice. On `gemm_bf16_n512_k512`, Claude Opus 4.8 switches to streaming mode and uses MOPA/ZA accumulation, reaching $0.94\times$ the expert implementation. Claude Sonnet 4.6 and DeepSeek-V4-Pro, in contrast, stay with older vector-style implementations and reach only $0.06\times$ and $0.05\times$. The same holds across kernels: nearly all final SVE solutions use SVE intrinsics, but only 5 of 7 (Opus) and 2 of 7 (Sonnet) final SME2 solutions use the ZA tiles (Appendix E). Appendix D.2 traces one representative SME2 optimization trajectory in detail.

Agents adapt more readily to incremental ISA extensions. By contrast, the move from SVE to SVE2 is much less disruptive. On the matched SIMD Loops kernels, Claude Opus 4.8, Claude Sonnet 4.6, and GLM-5.3 stay essentially level relative to the generation-specific expert references, with SVE2-to-SVE ratios of 1.07, 1.01, and 1.09; only DeepSeek-V4-Pro drops, to 0.78 (Table 4). The fixed-machine instruction ladder shows a similar pattern: Neon and SVE2 on Graviton4 reach $1.09\times$ and $1.10\times$ (Table 3). The SVE arm is lower ($0.96\times$) mainly because of two floating-point reduction loops, on which the agent kept writing SVE code that failed the correctness check and found a correct vectorization only after the 40-call budget; without these two kernels, the three arms differ by at most 8%. This is a milder form of the SME2 failures above: when agents cannot get a newer programming interface right, their kernels end up slower than plain Neon code. Agents therefore adapt more easily to incremental ISA extensions than to the programming-model shift introduced by SME2.

4.3 Agent Gains Depend on Expert Headroom

Across the 78 SVE kernels, median agent Speedup@40 decreases as the expert reference becomes stronger relative to unvectorized scalar code (Spearman $\rho = -0.68$; $\rho = -0.75$ within llama.cpp; Figure 3). Thus, large Speedup@40 values often reflect under-optimized reference paths rather than unusually strong agent performance. This is clear in llama.cpp RMSNorm: the reference is only $1.3\times$ faster than scalar, while agents beat it by a median $6.0\times$. Conversely, for q4_K_M GEMM and MoE, the expert implementations are already 25–29$\times$ faster than scalar, and agents reach only 0.57–$0.69\times$ of the expert.

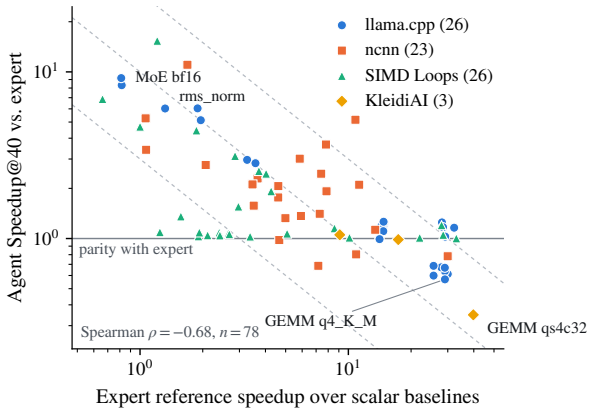


Figure 3: **Agents gain more when expert kernels are less optimized.** One point per kernel. x: expert speedup over the scalar reference; y: median Speedup@40 across harness–model cells. Dashed lines show constant agent speedup over scalar (3×, 10×, 30×).

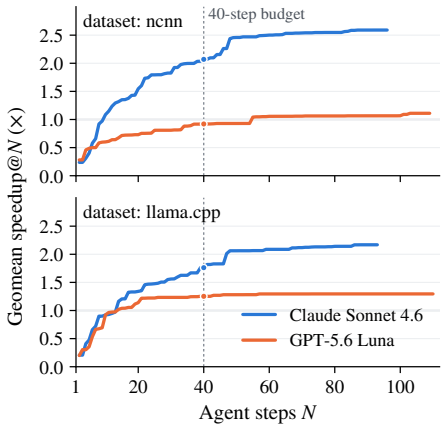


Figure 4: **Test-time scaling of agents.** Best-so-far Speedup@t against the expert reference as a function of the tool-call budget t. Most agent progress is within 40 steps.

4.4 What Improves Agent Optimization?

As shown in Figure 4, most of the improvement is realized within the first 40 tool calls, but budget beyond 40 still pays off: Claude Sonnet 4.6 is 23–25% higher at 100 calls than at 40. This additional gain, however, comes mainly from a few kernels undergoing a single structural rewrite between roughly 40 and 60 calls; past 60 calls the curves are largely converged.

Arm documentation does not improve performance on SVE and hurts it on SME2 (0.573 vs. 0.770). The documentation does not explain the streaming annotations that SME code requires in our harness (e.g., `__arm_locally_streaming`), so agents fail to compile SME code, conclude that SME is unsupported, and fall back to other ISAs earlier than without it.

Table 5: **Ablations** on the 32-kernel subset (nanobot + GPT-5.6-luna).

Setting	Speedup@40
SVE	
Control	1.074
+ Arm docs	0.997
+ cross-kernel notepad	0.974
− disassemble tool	0.999
No intrinsics	0.300
SME2	
Control	0.770
+ Arm docs	0.573

Some tools contribute little in practice. The cross-kernel notepad does not improve performance, while removing the disassembly tool does not lower performance. Inspection of agent trajectories shows that agents rarely use disassembly, while human experts frequently use it while designing advanced kernels. In contrast, forbidding SIMD intrinsics, so that agents must rely on compiler auto-vectorization, lowers performance from 1.074 to 0.300: the agents' gains come from hand-written vector code, not from restructuring the scalar code alone.

4.5 From Kernel Optimization to End-to-End Potential

To test whether kernel-level gains can translate into application-level improvements, we use Claude Code with Claude Fable 5.1 to optimize 11 quantized matrix-multiplication kernels for Qwen3.5-4B and 7 kernels for Llama-3.1-8B-Instruct (both Q4_K_M), served with llama.cpp on Graviton4, and substitute the resulting shape-specialized implementations into the inference stack.

Compared with original llama.cpp using its repacked i8mm fast path, the agent kernels speed up prefill by

Table 6: **End-to-end speedup from agent-optimized kernels.** Qwen3.5-4B and Llama-3.1-8B-Instruct (Q4_K_M) on Graviton4, relative to original llama.cpp.

	Qwen3.5-4B		Llama-3.1-8B-Instruct	
Threads	Prefill	Decode	Prefill	Decode
1	1.408	1.081	1.365	1.081
4	1.392	1.079	1.352	1.045
16	1.330	1.078	1.306	1.112

1.31–1.41× and decode by 1.05–1.11× across 1–16 threads for both models (Table 6). Perplexity rises by 0.6% on Qwen3.5-4B and is unchanged on Llama-3.1-8B-Instruct.

Gains come from adaptation to hardware. All 18 kernels use Graviton4's i8mm instructions together with shape-specific tiling, layout, and prefetching. Measured in isolation, individual kernels average 1.6× faster than llama.cpp's own repacked kernels. Unlike llama.cpp's general kernels, they are specialized to the target models' shapes, so this shows the potential of deployment-time specialization rather than general superiority over expert libraries. Appendix D.1 provides the complete setup and detailed results.

5 Related Work

Kernel optimization across hardware. Recent work evaluates whether language models and coding agents can generate efficient kernels for a given hardware target or programming interface. KernelBench and TritonBench study GPU kernel generation (Ouyang et al., 2025; Li et al., 2025), while MobileKernelBench and SimdBench extend to mobile CPUs and SIMD intrinsics (Zou et al., 2026; He et al., 2025). Other benchmarks extend kernel evaluation across architectures and devices (Li et al., 2026; Wen et al., 2025; Zang et al., 2026; Chen et al., 2025). These works primarily evaluate performance on a fixed target or a set of hardware targets. In contrast, KernelShift organizes hardware as a sequence of successive architectural extensions and evaluates matched kernel tasks across generations where coverage permits. This lets us study adaptation to new instructions and programming abstractions.

Coding agents for performance optimization. Recent coding agents improve programs through iterative execution and feedback (Yang et al., 2024; Wang et al., 2025). For kernel optimization, GPU Kernel Scientist and CudaForge use iterative search and hardware feedback, while AdaExplore combines execution-driven adaptation with diversity-preserving search (Andrews & Witteveen, 2025; Zhang et al., 2025; Du et al., 2026). More recent systems train kernel-optimization agents with reinforcement learning (Dai et al., 2026; Fu et al., 2026). These works show rapid progress in agentic kernel optimization. KernelShift provides both a controlled benchmark and a hardware-in-the-loop environment for testing whether such agents can adapt their optimization strategies as hardware evolves.

Automated program optimization. A broader line of work studies program optimization using learned cost models, reinforcement learning, and search (Chen et al., 2018; Zheng et al., 2020; Trofin et al., 2021; Cummins et al., 2022; Wei et al., 2025; Mankowitz et al., 2023; Fawzi et al., 2022; Novikov et al., 2025). However, automated optimizers can still fall short of expert implementations when performance depends on hardware-specific transformations (Mendis et al., 2019; Tschand et al., 2026). This gap motivates coding agents that can optimize kernels through interaction with real hardware.

6 Conclusion

We introduced KernelShift, a benchmark for studying how well coding agents optimize kernels as hardware evolves. Our results show that, on fixed workload sets, stronger agents can match or beat production kernels on established vector ISAs, but they struggle with programming-model shifts such as SME2. At the same time, the end-to-end speedups that agent-written kernels deliver on two LLMs highlight the potential of agent-driven kernel optimization in real deployments.

Limitations. Newer hardware is harder to evaluate because strong expert implementations are often not available, which limits how many kernels we can compare across generations; in particular, our SME2 evidence rests on the seven KleidiAI kernels that have an SME2 expert reference. Our experiments also focus on Arm CPUs, so it remains to be seen whether the same trends hold for other architectures such as x86 or GPUs. Finally, our kernel-level evaluation is single-core and does not cover all effects that may appear in multi-core deployment.

AI USE STATEMENT

AI agents are the primary subject of this study. We also used AI tools to assist with research code generation, benchmark data collection and synthesis, and wording refinement. All AI-generated code and benchmark instances, and paper refinements were reviewed and verified by the authors. The authors were responsible for the code, experiments, and manuscript.

ETHICS STATEMENT

KERNELSHIFT is constructed from open-source software, and all redistributed code follows the licenses of the original repositories: ncnn (BSD-3-Clause), llama.cpp (MIT), Arm SIMD Loops (BSD-3-Clause), and KleidiAI (Apache-2.0). The study does not involve human subjects.

REPRODUCIBILITY STATEMENT

The code for the benchmark and the dataset are available in the supplementary materials, including the scripts for evaluating the coding agents.

REFERENCES

Martin Andrews and Sam Witteveen. GPU kernel scientist: An LLM-driven framework for iterative kernel optimization. In *ES-FoMo Workshop at ICML*, 2025.

Anthropic. Claude Code. `https://github.com/anthropics/claude-code`, 2025. Accessed 2026-09-14.

Anthropic. Introducing Claude Fable 5.1 and Claude Mythos 5.1. `https://www.anthropic.com/claude-fable-and-mythos-5-1`, 2026a. Accessed 2026-09-25.

Anthropic. Introducing Claude Opus 4.8. `https://www.anthropic.com/news/claude-opus-4-8`, 2026b. Accessed 2026-09-21.

Anthropic. Introducing Claude Sonnet 4.6. `https://www.anthropic.com/news/claude-sonnet-4-6`, 2026c. Accessed 2026-09-21.

Arm. KleidiAI. `https://gitlab.arm.com/kleidi/kleidiai`, 2024. Apache-2.0. Accessed 2026-09-25.

Arm. SIMD Loops. `https://gitlab.arm.com/architecture/simd-loops`, 2025. BSD-3-Clause. Accessed 2026-09-14.

Arm. Arm C Language Extensions. `https://github.com/ARM-software/acle/releases/tag/r2026Q1`, 2026. ACLE 2026 Q1.

Siyuan Chen, Zhichao Lu, and Qingfu Zhang. Evolution of kernels: Automated RISC-V kernel optimization with large language models. *arXiv preprint arXiv:2509.14265*, 2025.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. CompilerGym: Robust, performant compiler optimization environments for AI research. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 92–105, 2022.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA agent: Large-scale agentic RL for high-performance CUDA kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

DeepSeek-AI. DeepSeek-V4 model card. https://fe-static.deepseek.com/chat/transparency/deepseek-V4-model-card-EN.pdf, 2026. Accessed 2026-09-21.

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

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

Matthew Ferrari, Philippe Tillet, Ahmed Ibrahim, Joe Gershenson, and Steve Coffey. How GPT-5.6 fuses frontier intelligence with frontier efficiency. OpenAI, https://openai.com/index/gpt-5-6-frontier-intelligence-efficiency/, July 2026.

Dayuan Fu, Mohan Jiang, Tongyu Wang, Dian Yang, Jiarui Hu, Liming Liu, Jinlong Hou, and Pengfei Liu. daVinci-kernel: Co-evolving skill selection, summarization, and utilization via RL for GPU kernel optimization. *arXiv preprint arXiv:2606.16497*, 2026.

Georgi Gerganov and ggml-org contributors. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp, 2023. Accessed 2026-09-16.

Yibo He, Shuoran Zhao, Jiaming Huang, Yingjie Fu, Hao Yu, Cunjian Huang, and Tao Xie. SimdBench: Benchmarking large language models for SIMD-intrinsic code generation. *arXiv preprint arXiv:2507.15224*, 2025.

HKUDS. nanobot: Ultra-lightweight open-source personal AI agent framework. https://github.com/HKUDS/nanobot, 2025. Accessed 2026-09-14.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. TritonBench: Benchmarking large language model capabilities for generating Triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, 2025.

Jie Li, Wenzhao Wu, Junqi Hu, Qinrui Zheng, Bowen Wu, Juepeng Zheng, Yutong Lu, and Haohuan Fu. CodegenBench: Can LLMs write efficient code across architectures? *arXiv preprint arXiv:2606.04023*, 2026.

Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin Riedmiller, Oriol Vinyals, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. Revec: Program rejuvenation through revectorization. In *Proceedings of the 28th International Conference on Compiler Construction (CC)*, 2019.

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

OpenAI. OpenAI Codex CLI. https://github.com/openai/codex, 2025. Accessed 2026-09-21.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *International Conference on Machine Learning (ICML)*, 2025.

Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 446–459, 2020. doi: 10.1109/ISCA45697.2020.00045.

Stefan Remke and Alexander Breuer. Hello SME! Generating fast matrix multiplication kernels using the Scalable Matrix Extension. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1443–1454, 2024. doi: 10.1109/SCW63240.2024.00185.

ShareGPT. ShareGPT_Vicuna_unfiltered. `https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered`, 2023. File ShareGPT_V3_unfiltered_cleaned_split.json. Apache-2.0. Accessed 2026-09-25.

Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2):26–39, 2017.

Tencent. ncnn: A high-performance neural network inference framework optimized for the mobile platform. `https://github.com/Tencent/ncnn`, 2017. Accessed 2026-09-14.

Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. MLGO: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808*, 2021.

Arya Tschand, Charles Hong, Julian Walker, Nina Cai, Shangkun Wang, Suvinay Subramanian, Sundar Dev, Vijay Janapa Reddi, Amir Yazdanbakhsh, and Sethu Sankaran. JAXBench: Benchmarking autonomous TPU kernel optimization. *arXiv preprint arXiv:2607.20466*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025.

Anjiang Wei, Tarun Suresh, Huanmi Tan, Yinglun Xu, Gagandeep Singh, Ke Wang, and Alex Aiken. SuperCoder: Assembly program superoptimization with large language models. *arXiv preprint arXiv:2505.11480*, 2025.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. MultiKernelBench: A multi-platform benchmark for kernel generation. *arXiv preprint arXiv:2507.17773*, 2025.

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

Peiyu Zang, Jian Tao, Jialing Zhang, Yichen Yuan, Wentao Zhang, Guang Liu, and Yonghua Lin. KernelGenBench: Can LLMs and agents write efficient kernels across operator sources and hardware platforms? *arXiv preprint arXiv:2607.27231*, 2026.

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

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

Zhipu AI. GLM-5.3. `https://huggingface.co/zai-org/GLM-5.3`, 2026. Accessed 2026-09-21.

Xingze Zou, Jing Wang, Yuhua Zheng, Xueyi Chen, Haolei Bai, Lingcheng Kong, Syed A. R. Abu-Bakar, Zhaode Wang, Chengfei Lv, Haoji Hu, and Huan Wang. MobileKernelBench: Can LLMs write efficient kernels for mobile devices? *arXiv preprint arXiv:2603.11935*, 2026.

A Benchmark Composition Details

A.1 Composition by source

KernelShift contains 85 kernel specifications across four source families: 23 from ncnn, 26 from llama.cpp, 26 from Arm SIMD Loops, and 10 from KleidiAI. The KleidiAI inventory comprises three SVE tasks and seven SME2 tasks, so the 85-kernel inventory is the union across hardware targets rather than a task set evaluated in full on every generation.

For each of the four sources, ncnn (Tencent, 2017), llama.cpp (Gerganov & ggml-org contributors, 2023), Arm SIMD Loops (Arm, 2025), and KleidiAI (Arm, 2024), Table 7 lists the operators it contributes, their precisions, and where the evaluated shapes and the expert reference come from.

Table 7: **Kernel sources in KernelShift.** Numbers in parentheses are kernel counts.

Source	Kernels	Precisions	Workload shapes	Expert reference	#
ncnn	conv2d (10), depthwise conv2d (8), pooling (5)	FP32, INT8 (w8a8ch)	6 vision models: ResNet50, MobileNetV3-Large, VGG16, GoogLeNet, AlexNet, SqueezeNet1.1	ncnn Arm (Neon) layers	23
llama.cpp	GEMM (15), MoE (6), attention: MHA, GQA, MLA prefill, MLA decode (4), RMSNorm (1)	BF16, q8_0, q4_K_M, FP32	5 LLMs on ShareGPT prompts: Qwen1.5-MoE-A2.7B, OLMoE-1B-7B, DeepSeek-V3, Llama-3.1-8B, Mistral-7B	ggml CPU kernels	26
Arm SIMD Loops	reductions and dot products (9), element-wise arithmetic (4), complex arithmetic (4), convolution and correlation (2), string and pixel processing (2), sorting, search, and conditional update (4), INT8 matrix–vector product (1)	FP16–FP64, INT8–INT64	edge-case and performance sizes	Arm's hand-written SVE version of each loop	26
KleidiAI	SVE: GEMM (2; FP32, INT4), conv2d 3×3 (1). SME2: GEMM (4; FP32, FP16, BF16, INT4), conv2d 3×3 (1), depthwise conv2d 3×3 (2)	FP32, FP16, BF16, INT4	edge-case and performance sizes	KleidiAI SVE and SME2 micro-kernels	3 + 7
Total					85

A.2 TASK SETS PER EXPERIMENT

Table 7 lists what the benchmark contains; Table 8 lists which kernels each experiment uses, on which machine, and against which reference. Only the main comparison uses the full SVE set; the other experiments use subsets chosen so that every arm or generation being compared has the same kernels and a valid reference.

Table 8: **Task set of each experiment.** Kernel counts are ncnn / llama.cpp / SIMD Loops / KleidiAI. G3, G4: Graviton3, Graviton4.

Experiment	Where	Machine	Harness, model	Kernels	Reference
Main comparison	Tab. 2	G3	all 14 pairs	23 / 26 / 26 / 3 (78)	expert
ISA ladder L1–L3	Tab. 3	G4	nanobot, GPT-5.6-luna	32-subset: 10 / 9 / 12 / 1	expert, same in every arm
ISA ladder L4–L6	Tab. 3	M4	nanobot, GPT-5.6-luna	KleidiAI SME2: 7	expert, same in every arm
KleidiAI SVE→SME2	Tab. 4	G3→M4	nanobot, 4 models	KleidiAI: 3 → 7	KleidiAI SVE → SME2
SIMD Loops SVE→SVE2	Tab. 4	G3→G4	nanobot, 4 models	SIMD Loops: 7	Arm SVE → SVE2 versions
Ablations, SVE	Tab. 5	G3	nanobot, GPT-5.6-luna	32-subset: 10 / 9 / 12 / 1	expert
Ablations, SME2	Tab. 5	M4	nanobot, GPT-5.6-luna	32-subset (SME2 version): 10 / 9 / 12 / 1	expert
Test-time scaling	Fig. 4	G3	nanobot, GPT-5.6-luna; Claude Sonnet 4.6	11 / 8 / 0 / 0 (19)	expert
End-to-end	App. D.1	G4	Claude Code, Claude Fable 5.1	11 Qwen3.5-4B and 7 Llama-3.1-8B-Instruct GEMMs	ggml; original llama.cpp

32-kernel subset. A dtype-balanced sample of the benchmark restricted to kernels with a valid expert reference on the target. ncnn: FP32 and w8a8ch versions of depthwise 3×3 and 5×5 (stride 1), conv2d 3×3 (stride 1) and 7×7 (stride 2), global average pooling, and 3×3 max pooling (stride 2). llama.cpp: GEMM BF16, q4_K_M and q8_0 ($N{=}K{=}2048$), GEMM q8_0 ($N{=}1408$), MHA, and MoE BF16, q4_K_M and q8_0 (OLMoE). SIMD Loops: `loop_001`, `002`, `003`, `024`, `027`, `035`, `037`, `105`, `108`, `113`, `126`, `127`. KleidiAI: FP32 GEMM, in its SVE version on Graviton and its SME2 version on M4.

SIMD Loops across generations. `loop_037`, `103`, `105`, `109`, `112`, `113`, and `219`, each scored against Arm's SVE implementation on Graviton3 and its SVE2 implementation on Graviton4. These are the only loops for which both the SVE and the SVE2 expert implementations are stable.

Test-time scaling subset. The ncnn kernels of the 32-kernel subset plus the w8a8ch 1×1 conv2d (stride 1), and the llama.cpp kernels of the 32-kernel subset except the q8_0 GEMM with $N{=}1408$, with the q8_0 MoE taken from Qwen1.5-MoE rather than OLMoE. The Claude Sonnet 4.6 curves use the main-comparison runs restricted to these 19 kernels.

End-to-end kernels. The 11 quantized GEMM shapes of Qwen3.5-4B, given as (N, K): q4_K at (1024, 2560), (2560, 4096), (2560, 9216), (4096, 2560), (8192, 2560), and (9216, 2560); q5_K at (2560, 4096) and (8192, 2560); q6_K at (1024, 2560), (248320, 2560), and (2560, 9216); and 7 quantized GEMM shapes of Llama-3.1-8B-Instruct, given as (N, K): q4_K at (1024, 4096), (4096, 4096), (4096, 14336), and (14336, 4096); q6_K at (1024, 4096), (4096, 14336), and (128256, 4096).

A.3 KERNEL INVENTORY

Most kernels in KERNELSHIFT are instances of a few operator families that vary in precision and in fixed operator attributes. Table 9 lists all 85 kernels in this compact form: each row is an operator family, and its variants expand to the listed number of kernels. Unless stated otherwise, convolution variants are written as kernel size and stride, GEMM variants as (N, K), and every kernel runs on SVE.

Table 9: **All 85 kernels by operator family.** $\times$: cross product; s: stride; SIMD Loops are given by loop number. Qwen: Qwen1.5-MoE-A2.7B; OLMoE: OLMoE-1B-7B.

Source	Family	Variants	#
ncnn	conv2d	{FP32, w8a8ch} $\times$ {1×1 s1, 1×1 s2, 3×3 s1, 3×3 s2, 7×7 s2}	10
	depthwise conv2d	{FP32, w8a8ch} $\times$ {3×3, 5×5} $\times$ {s1, s2}	8
	pooling (FP32)	global average; max 2×2 s2; max 3×3 s1; max 3×3 s2 with and without padding	5
llama.cpp	GEMM	{BF16, q8_0, q4_K_M} $\times$ {(1024, 2048), (1408, 2048), (2048, 1024), (2048, 1408), (2048, 2048)}; q4_K_M uses K=1536 in place of 1408	15
	MoE	{BF16, q8_0, q4_K_M} $\times$ {Qwen: 60 experts, top-4; OLMoE: 64 experts, top-8}	6
	attention (BF16)	MHA (Qwen); GQA, 32 query and 8 KV heads (Llama-3.1-8B, Mistral-7B); MLA prefill and MLA decode (DeepSeek-V3)	4
	RMSNorm (FP32)	hidden size 2048 (Qwen)	1
SIMD Loops	reductions, dot products	`001`, `002`, `003`, `004`, `008`, `024`, `033`, `105`, `113`	9
	element-wise arithmetic	`027`, `028`, `029`, `035`	4
	complex arithmetic	`037`, `109`, `110`, `112`	4
	convolution, correlation	`038`, `114`	2
	string, pixel processing	`103`, `108`	2
	sort, search, conditional	`120`, `124`, `126`, `127`	4
	matrix–vector	`219` (INT8)	1
KleidiAI	GEMM, SVE	FP32 (512, 512); INT4 weights with INT8 activations (256, 512)	2
	conv2d, SVE	FP32 NHWC 3×3 s1, 32→64 channels	1
	GEMM, SME2	{FP32, FP16, BF16} (512, 512); INT4 weights (256, 512)	4
	conv2d, SME2	FP16 NHWC 3×3, 32→64 channels	1
	depthwise conv2d, SME2	{FP32, FP16} NHWC 3×3 s1	2
Total			85

Inclusion criteria. A constructed kernel enters the benchmark only if it has a usable expert reference to measure the agent's relative performance. After a task is constructed, we build and run the expert implementation on the target machine and flag the kernel if (i) no expert implementation could be obtained from the source, (ii) the expert implementation fails to build or does not pass the correctness check on the task's workloads, or (iii) it is 2x slower than the compiler auto-vectorized version of the scalar starting point, in which case it does not represent expert performance. Flagged kernels are inspected and removed; the remaining kernels form the inventory in Table 9.

B TASK CONSTRUCTION

B.1 CONSTRUCTION SKILLS AND VALIDATION SCRIPTS

Task construction is carried out by a coding agent that follows written procedures, which we call *construction skills*. Each skill is a document that the agent loads on demand. It states what to extract from the request and given resource, the artifact to produce and its schema, the checks to run afterwards, and a list of known failure modes. The scripts are deterministic; the agent supplies the judgement the scripts cannot, such as which dimensions are fixed per kernel, how to call the source

library, and which workloads fill a coverage gap, and it runs the validation on the target machine. For the entire generation pipeline, we use Claude Code with sonnet-5 as the base model.

Four skills cover the pipeline of Section 2.1. `gen-definition` writes the kernel specification (Figure 5), `gen-workload` collects its workloads (Figure 6), `gen-candidate-solution` renders the scalar starting implementation (Figure 7), and `gen-baseline-solution` wraps the source library's kernel as the expert reference (Figure 8). The figures show condensed versions; the complete skill documents are released with the benchmark. The skills stop for human input only at a few fixed points, described next.

Human involvement. Humans make three kinds of decisions. They select kernels, approving each definition that `gen-definition` proposes and, for large libraries such as KleidiAI with hundreds of micro-kernels, choosing a representative subset. They approve the workloads of kernel libraries (Arm SIMD Loops and KleidiAI), which the agent proposes because there is no model execution to trace. They review each expert reference after its first evaluation, since a correct wrapper can still underperform the library; an early KleidiAI wrapper, for example, allocated and freed buffers on every call. If such a wrapper cannot be fixed, the kernel is excluded (Appendix A.3). The scalar starting implementations need no human input.

Construction Skill: `gen-definition`

Task Description
Design and write a Definition JSON for a new operator. This skill writes the structural schema only; concrete tensor sizes are added afterwards by `gen-workload`. The reference function inside the definition is a correctness oracle, not a workload.
What to Extract from the Request
* Op name and what it computes (enough to write a reference function)
* Axes: which are fixed per kernel configuration (`const`) and which vary per workload (`var`)
* Input and output tensors: names, shapes (as lists of axis names), and dtypes
* Constraints expressing derived output axes in terms of other axes
* A reference implementation in NumPy or PyTorch

If the request is underspecified in any of these areas, ask before generating.
Workflow
0. If the op comes from a real model, extract its shapes from the Hugging Face `config.json` or from torchvision forward hooks (no weights are downloaded), and record the source as a `model:<name>` tag.
1. Read one or two existing definitions of the closest operator family.
2. Elicit the design from the user until it is unambiguous, including which framework supplies the expert reference (`baseline-solution:<framework>` tag). **Always ask; do not infer it from the model or the dtype.**
3. Present the complete definition and its output path. **Wait for explicit confirmation before writing.**
4. Write the JSON, then check that every axis used in a shape or constraint is declared, every `const` axis has an integer value, every derived `var` axis has a constraint, and the reference function uses integer literals rather than axis names.
5. Verify that the benchmark loads the definition; diagnose and fix any error before declaring success.
Conventions
* Only `var` axes appear in workloads; `const` axes live in the definition.
* Precision variants are separate definitions with the scheme in the name: `fp32`; `q8_0` (llama.cpp, block size 32, int8 values with fp16 block scales, fp32 output); `w8a8ch` (ncnn, per-tensor activation scale, per-channel weight scales, requantized int8 output).
[. . . known failure modes and their fixes omitted . . .]

Figure 5: **Construction skill for kernel definitions**, condensed from the skill document given to the construction agent (about 1,900 words). Command lines, file paths, and the table of known failure modes are omitted.

Construction Skill: `gen-workload`

Task Description
Add workloads to the definitions of an operator family. The primary path is always automated collection; manual additions are for targeted gap-fills only.
Workflow
1. Identify the operator family and its collection path:
* Vision kernels: hook the matching module of torchvision ResNet50 and MobileNetV3-Large over 11 input resolutions (96 to 1024, plus 360×640 and 480×640).
* LLM kernels: run llama.cpp with a GGUF model on 200 ShareGPT prompts on a Graviton instance and record the operator shapes.
* Neither: go to manual addition (step 4).
Captured shapes are matched to the definition by its own `const` axes, so a new definition in a supported family needs no script change.
2. Run the collector, previewing with a dry run first.
3. The collected pool is filtered automatically: keep at most two copies of any identical axis combination; seed one workload per distinct value of a batch axis (`C_in` for convolution, `C` for depthwise and pooling, `M` for GEMM, `n_tokens` for MoE); fill the remaining slots greedily by farthest-point selection in normalized axis space, up to 20 workloads.
4. For manual additions, **present a table of the planned workloads and their rationale, and wait for explicit confirmation before writing.** Check coverage: at least one channel count that is not a multiple of common SIMD widths; 15–20% of workloads with an odd spatial dimension; for attention, the decode point $M{=}1$ at the maximum context length.
5. Verify the workload count and that the benchmark loads them.
Conventions
* A workload row holds only the `var` axes; its ID is a deterministic hash of the definition and the axes.
* `w8a8ch` definitions receive an activation scale drawn from a bounded range and seeded by the definition name, so it is reproducible across reruns.
[... command lines, per-family axis tables, and known failure modes omitted ...]

Figure 6: **Construction skill for workloads**, condensed from the skill document (about 1,500 words).

B.2 WORKLOADS

Workloads for ncnn and llama.cpp kernels come from model executions: operator shapes observed in ResNet50 and MobileNetV3-Large at 11 input resolutions, and in llama.cpp inference on 200 ShareGPT prompts (ShareGPT, 2023) of 10–1500 words with a context of up to 4096 tokens. The collected shapes are deduplicated and subsampled to about 20 per kernel (Figure 6). Arm SIMD Loops and KleidiAI have no model execution to trace; their workloads are proposed by the agent and approved by humans, and combine small edge-case sizes for correctness with large sizes for performance.

Only shapes are kept from the executions. Input tensors are generated deterministically based on each workload's ID, and quantization scales are drawn from fixed ranges, so every evaluation of a workload sees the same inputs. Across the 85 kernels, the benchmark has 1,288 workloads.

B.3 CORRECTNESS CRITERIA

An output element matches if it is within both an absolute and a relative tolerance of the reference value; positions where the reference itself is not finite are ignored. A workload passes if the fraction of matching elements reaches the required match ratio, and a candidate is correct only if it passes every workload. Kernels whose error is dominated by quantization or low-precision accumulation are instead checked with the signal-to-quantization-noise ratio, $\mathrm{SQNR} = 10\log_{10}\left(\sum_i r_i^2 / \sum_i (y_i - r_i)^2\right)$, against a minimum. Table 10 lists the thresholds. The MLA decode and FP16 depthwise thresholds are loosened so that their own expert references pass: the BF16 MLA decode reference deviates from the Python reference by up to 2.9×10^{-2}, and the FP16 depthwise reference matches only 96.7–98.2% of elements at the default tolerance.

Construction Skill: `gen-candidate-solution`

Task Description
Generate the scalar starting implementation for the definitions of an operator. Each solution embeds three sources: a header holding the harness contract with the definition's `const` axes baked in as `constexpr`, a binding that implements the common entry symbol `kernelshift_entry_<op_type>`, and a kernel file with plain scalar loops, which is the file the optimizing agent later replaces. Everything is derived from the definition; there is no configuration file.

What to Extract from the Request
* Op type, and whether to target one definition or all of them
* Variant: plain FP32 or a quantized scheme, read from the definition's input dtypes
* Mode: a new operator or variant, or regeneration of existing solutions

Workflow
0. Check the definition's `baseline-solution` tag and generate only for definitions of the matching source.
1. Classify the definition: `const` axes become `constexpr`; `var` axes become entry parameters; a scalar input is baked only if it has the same value in every workload of the definition, which is checked against the workload file, and the definition is skipped otherwise.
2. Derive the entry signature from the definition: tensor pointers in declaration order with the output second, each typed by its declared dtype, followed by the `var` dimensions deduplicated across all tensor shapes in first-seen order.
3. Write the header, binding, and scalar kernel templates. For a quantized kernel, match the definition's Python reference: accumulate in 64-bit integers, then apply the dequantization scales and bias in double precision.
4. Generate the solutions (dry run first), then compile and run each on every workload on the target machine.

Rules
* The entry point always names the binding, never the kernel file.
* Only a `const` axis may become a compile-time constant; a `var` axis is always a run-time parameter.

[... command lines, template file layout, and known failure modes omitted ...]

Figure 7: **Construction skill for the scalar starting implementation**, condensed from the skill document (about 1,800 words).

Table 10: **Correctness thresholds.** Kernels not listed use the default.

Kernels	Absolute tol.	Relative tol.	Match ratio
Default (convolution, pooling, RMSNorm, SIMD Loops)	10^{-3}	10^{-3}	1.00
GEMM	2×10^{-3}	10^{-2}	0.98
MoE	10^{-2}	5×10^{-2}	0.95
MHA	10^{-3}	10^{-3}	0.98
GQA, MLA prefill	5×10^{-3}	10^{-3}	0.98
MLA decode	3×10^{-2}	10^{-2}	0.98
KleidiAI FP16 depthwise conv2d	10^{-3}	10^{-3}	0.95
	Minimum SQNR		
llama.cpp q4_K_M GEMM and MoE; q8_0 MoE	20 dB		
KleidiAI INT4 GEMM (SVE)	30 dB		
KleidiAI INT4 GEMM (SME2)	40 dB		
KleidiAI BF16 GEMM (SME2)	45 dB		

C Evaluation Infrastructure

C.1 Target machines

For each run, the control plane provisions a dedicated target machine (in AWS us-west-2 for Graviton, and a `mac-m4.metal` host for M4), evaluates the tasks sequentially with retries on failure,

Construction Skill: `gen-baseline-solution`

Task Description
Generate the expert-reference solution for the definitions of an operator by wrapping the source library's kernel. The solution has the same three sources as the starting implementation, but its kernel file delegates to the library (for ncnn, the Arm layer classes linked from the static library). One generator serves every source library; a new library is described by parameters, not by a new script.
What to Extract from the Request
* Op type and scope (one definition or all definitions tagged for this library)
* Library: if it is not an existing one, this is new-library setup; **confirm the scope with the user**
* Mode: a new operator or regeneration after the definitions changed
Workflow
1. Nothing needs registering: definitions are discovered from the definition directory, and every template value is derived from the definition itself.
2. Learn the library API for this operator by **reading the library's real headers rather than assuming it from another operator**. Several layer fields are not initialized by the constructor; set every field explicitly. Use a single thread and create the layer pipeline before calling it.
3. Write the header, binding, and kernel templates. Bake only `const` axes and workload-invariant scalars; derive every `var` dimension at run time from the tensor shapes. A template with an unresolved placeholder is the wrong template for that definition, and the definition is skipped.
4. Generate the solutions, verify that the benchmark loads them, then collect the baselines on the target machine to confirm that they compile and pass correctness.
Rules
* For int8 variants, the library's scale fields are the reciprocal of the definition's dequantization scales; invert them, or the output is silently wrong.
* Copy library tensors row by row or channel by channel, since rows and channels can carry alignment padding.
[... command lines, per-layer API notes, and known failure modes omitted ...]

Figure 8: **Construction skill for the expert reference**, condensed from the skill document (about 2,800 words). The omitted per-layer notes document initialization pitfalls of individual ncnn layer classes.

and collects each task's trajectory; the evaluator is reachable through an SSH tunnel. Table 11 lists the environment of each hardware generation. No hardware documentation is given to the agent in the main runs; the documentation ablation adds the Arm Neoverse V1/V2 optimization guides and an SVE intrinsics summary on SVE, and SME programming notes on SME2. On M4, performance

Table 11: **Evaluation environment per hardware generation.** Counter feedback lists the counters returned by *evaluate*; reported speedups use wall-clock time.

Gen.	Machine	Core, vector width	OS / compiler	Target flag	CPU pinning	Counter feedback
SVE	Graviton3, AWS `c7g.xlarge`	Neoverse V1, 256-bit SVE	Ubuntu 22.04, clang-18	`-march=armv8.2-a+sve`	CPU 0	cycles, instructions, cache misses
SVE2	Graviton4, AWS `c8g.xlarge`	Neoverse V2, 128-bit SVE2	Ubuntu 22.04, clang-18	`-march=armv9-a+sve2`	CPU 0	cycles, instructions, cache misses
SME2	Apple M4, AWS `mac-m4.metal`	512-bit, streaming SVE only	macOS, Apple clang	`-mcpu=apple-m4`	none	none

and efficiency cores cannot be selected separately, so comparisons between Neon and SME2 use the same M4 host; cross-generation comparisons use only shared kernels.

C.2 INSTRUCTION-SET LADDERS

Table 12 gives the configuration of each arm of the instruction-set ladder experiment (Table 3). On Graviton4, an arm is set by the compile target alone, since intrinsics of a newer extension do not compile under an older target. On Apple M4, all arms compile for the M4, and no compiler flag separates streaming SVE from ZA, because enabling SME also enables ZA. The Neon-only and streaming-SVE arms are therefore enforced by rejecting disallowed constructs in the submitted source and disallowed instructions in the disassembly of the compiled kernel. All arms compile with `-O3`, and within a machine every arm is scored against the same expert reference, so differences between arms reflect only the instructions available to the agent. The task set of each ladder is listed in Table 8.

Table 12: **Configuration of the instruction-set ladder arms** (Table 3).

Arm	Machine	Compile target	Additionally rejected
L1: Neon only	Graviton4	`-march=armv8-a`	— (SVE intrinsics do not compile)
L2: + SVE	Graviton4	`-march=armv8.2-a+sve`	— (SVE2 intrinsics do not compile)
L3: + SVE2	Graviton4	`-march=armv9-a+sve2`	—
L4: Neon only	Apple M4	`-mcpu=apple-m4+nosme`	Source: SVE and SME headers and types, streaming attributes, `smstart`. Binary: `smstart`, SVE and ZA/ZT0 registers.
L5: + streaming SVE	Apple M4	`-mcpu=apple-m4`	Source: ZA/ZT0 attributes and intrinsics, `smstart za`. Binary: `smstart` without `sm`, outer products (`MOPA`/`MOPS`), ZA/ZT0 registers.
L6: + ZA (full SME2)	Apple M4	`-mcpu=apple-m4`	—

C.3 AGENT INTERFACE, HARNESSES, AND PROMPTS

What the agent sees. For each task, the agent receives the kernel specification in code form: the definition name and operator type, including the constant attributes (e.g., kernel size and stride, or N and K of a GEMM), and the scalar starting implementation. It does not receive the workloads: the values of the variable dimensions (e.g., channel counts, spatial sizes, or token counts), the number of workloads, and the input tensors are never shown. The kernel receives the variable dimensions as run-time arguments, so a candidate must be correct for every size that the specification allows. Feedback is aggregated over the hidden workloads: *evaluate* reports the maximum errors, the geometric-mean speedup, and hardware counter averages where available, and after a failure only the identifier of the first failing workload and an error message. The agent also sees the results of its own earlier calls. Every *compile*, *evaluate*, and *disassemble* call counts toward the 40-call budget, a run counts as complete only after at least 15 calls, and the best correct candidate is retained.

The study uses 14 harness–model combinations, the rows of Table 2: Claude Code with the two Claude models, Codex with the two GPT-5.6 models, the tool-call loop with both pairs, and nanobot with all six models, so DeepSeek-V4-Pro and GLM-5.3 run only with nanobot.

All harnesses run without human input and receive the same task skill. Claude Code runs in a fresh, empty working directory with shell and web tools disabled and auto-memory off, and Codex runs in an empty workspace; both otherwise use their default settings. nanobot uses temperature 0.1, reasoning effort `xhigh`, and a 200k-token context, with shell and web tools disabled and file access limited to the task's workspace. The tool-call loop uses temperature 0.2 (none for Claude Opus, which rejects it). In long runs, the tool-call loop replaces earlier kernel versions with a summary of the versions that passed, and nanobot compacts older tool results when its context fills up.

Figure 9 shows the prompt given to the agent and Table 13 the tools it can call.

Agent Prompt

System (kernel-session skill, condensed):

Ground Rules

* Work on one definition at a time; *evaluate* and *disassemble* must name the definition and version of your last *compile*.
* Stay on the target ISA; do not fall back to an older one.
* Never use OpenMP or other parallelization. Kernels always run on the remote target, which has every dependency installed.
* Use *disassemble* to check whether SIMD is really helping and to find bottlenecks.

Workflow

0. Call *check_progress* to look up the best result so far. If one exists, compile and evaluate it first, then continue from it.
1. Read the `reference-scalar-kernel.cpp` resource, compile it unchanged (this becomes v1), and evaluate it to record the starting point.
2. Iterate: compile an attempt; evaluate it (correctness and timing in one call; the best version is saved automatically); disassemble when IPC is low or the speedup is unexpectedly poor. Re-read earlier versions and the trajectory instead of re-evaluating them.

Metrics from evaluate

* `max_absolute_error`, `max_relative_error`; `time_speedup_geomean` and `cycle_speedup_geomean` against the expert baseline (geometric mean over workloads); `ipc_mean`; `cache_misses_mean`.
* On failure, `failed_workload` and `log` say which workload failed and why.

User (task prompt):

Optimize the "`<definition>`" kernel definition (dataset: `<source>`, baseline solution source: `<expert>`) in ISA `<isa>`. You must spend at least 15 tool calls but not exceed 40 tool calls to explore genuinely different optimization attempts before you are allowed to submit. Once you hit that ceiling, stop iterating and submit your best version immediately, since every iteration spends real model API budget. Follow the ground rules and workflow in your system prompt.

Figure 9: **Agent prompt.** The skill (about 1,100 words) is condensed; the task prompt is verbatim.

Table 13: **Agent tools**, condensed from their schemas. *definition* and *version* must match the last *compile*; every *compile*, *evaluate*, and *disassemble* call counts toward the 40-call budget.

Tool	Arguments	What it does	Returns
`check_progress`	`definition`	Looks up the best result so far on this definition.	Its version, metrics, and code.
`compile`	`definition`, `code`	Compiles the agent's `kernel.cpp`; the harness and binding files are added automatically.	A version number, or the compiler error.
`evaluate`	`definition`, `version`	Checks correctness on all workloads, stopping at the first failure; if correct, times the kernel. A new best result is saved immediately.	Errors, speedups, and counters; or the failing workload's ID and a log.
`disassemble`	`definition`, `version`, optional `fn`	Disassembles the kernel entry symbol, or `fn`.	Up to 300 lines of AArch64 assembly.

C.4 Measurement protocol and integrity

Each workload is first run for correctness and then timed in 50 samples after a warmup; the primary metric is the minimum sample. On Apple M4, where many kernels run too briefly to be timed accurately, each sample repeats the kernel until it lasts at least 10 ms, and a candidate uses the same repetition count as the baseline it is compared against. On Linux, the timed process is pinned to one CPU; on M4, performance and efficiency cores cannot be selected. When available, hardware counters are also collected. Expert baselines are measured on the same machine before agent evaluation. Per-workload speedup is expert-reference time divided by candidate time, and task speedup

is the geometric mean over workloads. Each *evaluate* call runs in a fresh subprocess and is stopped if checking and timing all workloads takes longer than 1200 s.

Candidate execution is isolated in a subprocess. Across three seeds of nanobot with GPT-5.6-luna on Graviton4, on the 31 kernels of the 32-kernel subset other than the KleidiAI kernel, Speedup@40 varies by 4.5%, and per-kernel Speedup@40 by a median 9.7%. Re-timing the same fixed code varies by only 0.2%, so run-to-run variation comes from agent stochasticity rather than from measurement.

D Case Studies

D.1 End-to-end optimization with agent-written kernels

Setup. The main experiments measure adaptation one kernel at a time. To test whether these gains transfer to an application, we test two popular open-weight models, Qwen3.5-4B and Llama-3.1-8B-Instruct (both Q4_K_M), served with llama.cpp on Graviton4. 11 individual quantized matrix-multiplication kernels from the Qwen model and 7 matrix-multiplication kernels from the Llama model are turned into benchmark tasks. We substitute the resulting agent-written kernels into the inference stack and compare against original llama.cpp on the same hardware.

Model selection. These two models are compared to test whether the result depends on the architecture. Qwen3.5-4B is chosen due to its recency and hybrid architecture with both Gated DeltaNet and attention, so the model uses more kinds of kernels. Llama-3.1-8B-Instruct is chosen to test a different vendor and because it is a standard dense transformer, and widely used for CPU inference. We use 4-bit quantization for both, as it is the standard for CPU inference in llama.cpp. These models are small enough to measure reliably.

Table 14: **End-to-end speedup and perplexity for Qwen3.5-4B** (Claude Code + Claude Fable 5.1), relative to original llama.cpp.

Threads	Prefill	Decode
Agent kernels		
1	1.408	1.081
4	1.392	1.079
16	1.330	1.078
Original without repack		
1	0.485	0.748
4	0.493	0.738
16	0.544	0.868
Model	**Perplexity**	**Δ**
BF16	9.384	−1.3%
Q6_K	9.436	−0.7%
Q4_K_M	9.503	—
Q4_K_M + agent	**9.562**	+0.6%
Q3_K_M	9.846	+3.6%

Table 15: **Prefill speedup by prompt length and micro-batch** at 16 threads, relative to original llama.cpp.

Setting	Qwen3.5-4B	Llama-3.1-8B
Prompt length		
512	1.377	1.393
2048	1.350	1.322
4096	1.320	1.255
8192	1.279	1.184
Micro-batch size		
1	1.086	1.090
2	1.438	1.636
8	1.339	1.524
64	1.090	0.989
512	1.371	1.397

End-to-end performance and quality. Table 14 reports the speedup across different thread counts and quantizations relative to llama.cpp for Qwen3.5-4B, and Table 15 reports speedup across prompt length and micro-batch size for both models. For Qwen3.5-4B, the adapted kernels speed up prefill by $1.33\times$ at 16 threads and by 1.39–$1.41\times$ at 1–4 threads. Decode improves by approximately $1.08\times$, consistent with its greater sensitivity to memory bandwidth and to operators that are not replaced. Perplexity increases by 0.6% relative to the Q4_K_M model, slightly less than the 0.7% gap between Q6_K and Q4_K_M in this evaluation. For Llama-3.1-8B-Instruct, the agent kernels speed up prefill by 1.31–$1.37\times$ and decode by 1.05–$1.11\times$ across 1–16 threads (Table 6), and perplexity is unchanged (7.879 in both builds).

Serving regimes. The prefill gain persists across the evaluated serving conditions (Table 15). On Qwen, it declines from $1.38\times$ at 512 prompt tokens to $1.28\times$ at 8192 tokens, as the share of work in unreplaced operators increases. On Llama, the gain falls faster with prompt length, from $1.39\times$ to $1.18\times$, since attention runs in all of Llama's layers. Across micro-batch sizes, the largest measured gain is $1.44\times$ for Qwen and $1.64\times$ for Llama at a batch size of 2, while batch size 1 is close to the decode regime at $1.09\times$.

Adaptation to the hardware. All 18 kernels (11 for Qwen3.5-4B and 7 for Llama-3.1-8B-Instruct) use Graviton4's i8mm matrix-multiply instructions. The agent enables these instructions per function and combines them with shape-specific tiling, layout, and prefetching. Measured alone, agent kernels are $1.63\times$ faster for Qwen and $1.62\times$ for Llama than llama.cpp's repacked kernels, at similar accuracy. Each generated kernel serves one fixed shape, whereas a general-purpose library implementation must cover a broader shape distribution. We note that this is not a general claim that agents out-engineer expert libraries end-to-end, and rather demonstrates the potential of hardware-specific kernel generation.

D.2 Learning SME2 at test time

Table 16 lists every counted tool call of one run: nanobot with Claude Opus 4.8 on KleidiAI's BF16 GEMM on Apple M4, the run behind the $0.94\times$ result in Section 4.2.

The agent starts from the scalar implementation ($0.004\times$ the KleidiAI SME2 kernel) and targets the matrix engine directly, planning a `BFMOPA` outer-product kernel that accumulates into the ZA array. Its first attempt does not compile because it gathers operands inside streaming mode, where gather loads are unavailable, so the agent packs both operands to BF16 in non-streaming code. The next two kernels run but are incorrect, until the agent works out that the governing predicates of `BFMOPA` are defined over the 16-bit source elements rather than the 32-bit outputs. Six calls thus pass before the first correct SME2 kernel ($0.37\times$). Blocking the output over all four FP32 ZA tiles then more than doubles the speedup ($0.80\times$). The remaining calls are ordinary performance engineering: the agent finds that packing the operands, not the outer products, limits performance, and cache-blocked Neon packing brings the kernel to $0.935\times$ at call 29.

On the same kernel, Claude Sonnet 4.6 never enters streaming mode or uses ZA; its versions stay with Neon code and end at $0.06\times$.

Table 16: **Tool calls of one SME2 run** (KleidiAI `gemm_bf16_n512_k512`, nanobot with Claude Opus 4.8, Apple M4). Speedups are relative to the KleidiAI SME2 kernel.

Calls	What the agent does	Result
1–2	compile and evaluate the given scalar code	0.004×
3	first `BFMOPA` kernel, operands gathered inside streaming mode	compile error
4–7	pack operands to BF16 outside streaming mode; one ZA tile; fix load predicates	incorrect (twice)
8–9	16-bit governing predicates for `BFMOPA`	0.370×
10–11	2×2 blocking over all four FP32 ZA tiles	0.799×
12–20	Neon packing of B, disassembly, alternative packing of A, K unrolling	1 compile error; 0.507–0.799×
21–25	multi-slice ZA reads in the output store	1 compile error; 0.838–0.839×
26–29	cache-blocked Neon packing of A, wider packing of B (best)	**0.935**×
30–35	wider loads, branch hoisting, loop reordering	0.925–0.935×
36	one more packing variant, compiled but not evaluated before the run ends	—

E Use of Target-Specific Intrinsics

We check whether agents actually use the instructions of the target extension. For each run, the final solution is the version whose result is reported as Speedup@40. A solution counts as using SVE if it calls at least one SVE intrinsic, and as using SME2 if it calls at least one ZA intrinsic (names

containing `_za`); on SME2, the solutions that use ZA are exactly those that enter streaming mode. Table 17 reports the nanobot runs of the two Claude models. Nearly all final SVE solutions use SVE intrinsics, but on SME2 only 5 of 7 (Claude Opus 4.8) and 2 of 7 (Claude Sonnet 4.6) final solutions use the matrix tiles; the others never enter streaming mode, consistent with the drop from SVE to SME2 in Table 4.

Table 17: **Final solutions that use intrinsics of the target extension** (nanobot): SVE intrinsics on SVE (78 kernels), ZA intrinsics on SME2 (7 KleidiAI kernels).

Target	Claude Opus 4.8	Claude Sonnet 4.6
SVE	78/78 (100%)	74/78 (95%)
SME2	5/7 (71%)	2/7 (29%)

F Contamination Statement

Because ncnn, llama.cpp, SIMD Loops, and KleidiAI are public, expert code may exist in models' training data. However, on most ncnn, llama.cpp, and SIMD Loops kernels, agents exceed the expert code rather than reproducing it, and use intrinsics not in library code and different algorithm structures. Further, the task setup does not match the library: agents start from portable scalar code with benchmark-specific interfaces. Finally, KleidiAI's SME2 expert kernels are also public, yet agents fall far short of them.