RvvCoder: ISA-Grounded LLMs for High Performance RISC-V Vector Kernel Generation

Anonymous authors
Paper under double-blind review

Abstract

Open RISC-V hardware is spreading rapidly, yet every new vector core needs efficient kernels, and hand-writing RISC-V Vector (Rvv) assembly is costly. Rvv is vector-length agnostic, so a kernel must stay correct across vector lengths and fast on each core, which challenges both compilers and large language models (LLMs). We formalize Rvv kernel generation under instruction-set architecture (ISA) and hardware constraints and propose RvvCoder. RvvBench pairs RvvCoT, a specification-guided corpus, with a 38-operator suite verified across vector lengths and timed on real boards. RvvCoder learns ISA rules by supervised fine-tuning, then improves by reinforcement learning on kernels executed in QEMU and on hardware, where Decomposed Execution Verification (DEV) rewards partial progress. RvvCoder reaches 59% ER@10 correctness versus 13% for GPT-5.5, and its verified kernels run up to $1.49\times$ faster than the strongest applicable baselines on two Rvv boards, confirming the value of hardware feedback.

1 Introduction

Efficient kernels underpin scientific computing, signal processing, vision, and AI inference (Pirova et al., 2026; Li et al., 2023; Assir et al., 2021). Open RISC-V hardware is expanding rapidly (Lee et al., 2016), yet each core needs tuned kernels. We study generating correct, efficient RISC-V Vector (Rvv) assembly for real processors (Figure 1) (Perotti et al., 2022). Unlike fixed-width AVX or NEON, Rvv is vector-length agnostic: the active length depends on remaining work, vector configuration, and hardware capacity at run time. A kernel must stay correct across vector lengths and fast on each core, which makes generation valuable but demanding.

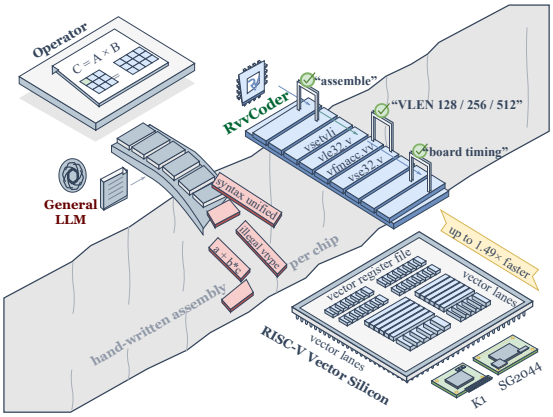


Figure 1: An illustration of RvvCoder.

Existing approaches are partial. Expert libraries like OpenBLAS Rvv and llama.cpp deliver strong kernels but require manual work per new operator (Xianyi et al., 2012; Liu et al., 2025). GCC and LLVM auto-vectorization is portable but conservative (Nuzman & Henderson, 2006; Lattner & Adve, 2004), and TVM searches only predefined schedules (Chen et al., 2018). Large language models (LLMs) generate CUDA, Triton, and accelerator kernels (Ouyang et al., 2025; Li et al., 2026; Woo et al., 2026; Cao et al., 2026b), but they rely on higher-level abstractions and do not transfer to vector-length-agnostic assembly, where frontier LLMs solve few tasks.

Applying LLMs to Rvv exposes three open problems. **(i) An undefined target:** Rvv generation couples ISA legality, calling conventions, cross-length correctness, and speed, yet no formulation states them jointly. **(ii) Scarce supervision and evaluation:** public Rvv data are rare, models leak syntax from other ISAs, and no benchmark checks both vector-length-agnostic correctness and real-hardware speed. **(iii) Sparse execution feedback:** legal instructions can form incorrect kernels,

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

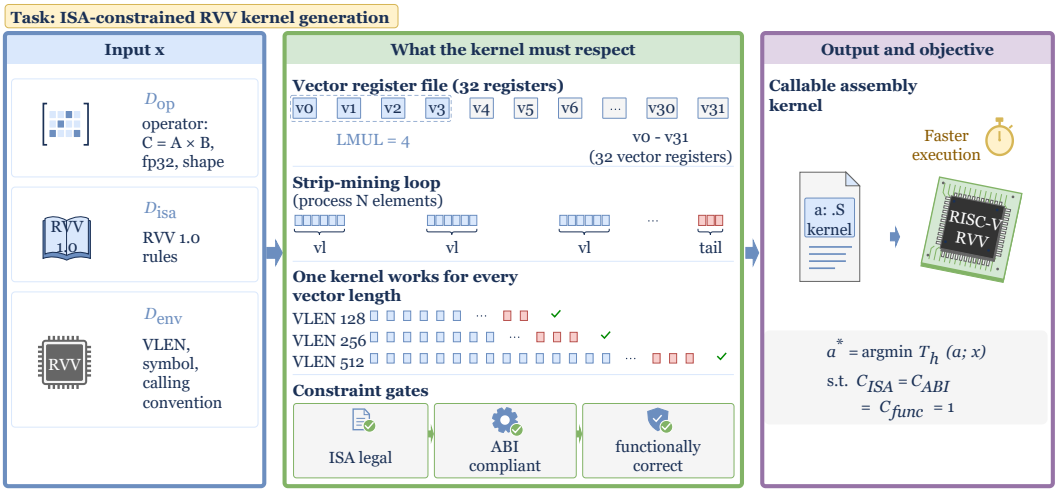


Figure 2: Overview of the Rvv task: from operator, ISA, and environment specifications, generate a callable kernel that meets register, strip-mining, and cross-VLEN constraints at minimal runtime.

and whole-kernel pass or fail rarely separates candidates. This raises our question: ***How can LLMs learn to generate ISA-compliant, functionally correct, and efficient Rvv assembly?***

To address these problems, we propose RvvCoder, an ISA-grounded LLM framework coupling domain supervision with hardware feedback. **ISA-Constrained Rvv Task:** To fix the target, we formalize synthesizing callable assembly from operator, ISA, and environment specifications (Figure 2), requiring ISA/ABI legality and cross-length correctness before minimizing runtime. **RvvBench:** To supply supervision and evaluation, specification-guided synthesis builds RvvCoT, a four-layer corpus of ISA rules, verified kernels, repairs, and performance reasoning, paired with a suite measuring compilation, correctness, and hardware efficiency. **RvvCoder:** To densify feedback, RvvCoder first learns ISA and ABI rules, then executes its own kernels in QEMU and on physical boards, where Decomposed Execution Verification (DEV) rewards partial progress.

RvvCoT spans four supervision layers, and the RvvBench suite covers 38 operators at three levels, scored by compilation rate, execution correctness over 20 candidates, and real-hardware speedups on two physical Rvv boards. RvvCoder reaches 59% Avg. ER@10, versus 13% for GPT-5.5, and 49.6% on the 28 Level-2/3 operators, versus 8.0%. Its verified kernels run faster than the strongest applicable baselines on every measured operator, by $1.27\times$ and $1.17\times$ on average and up to $1.49\times$ on the two boards, confirming that execution feedback yields correct and efficient kernels.

2 Related Work

RISC-V Vector Optimization. GCC/LLVM auto-vectorization can be conservative (Nuzman & Henderson, 2006; Lattner & Adve, 2004); TVM searches manually designed schedules (Chen et al., 2018); OpenBLAS Rvv and llama.cpp provide expert performance with limited coverage (Xianyi et al., 2012; Liu et al., 2025); and deep RL discovers faster sorting routines in assembly (Mankowitz et al., 2023). CUDA/Triton benchmarks standardize evaluation (Ouyang et al., 2025; Li et al., 2025b; Wen et al., 2025; Lange et al., 2025; Han et al., 2026; Lin et al., 2026), but Rvv assembly lacks training data and benchmarks that combine vector-length-agnostic correctness with real-hardware performance. RvvBench and RvvCoder address this gap directly for real Rvv hardware.

LLM-based Kernel Generation. LLMs optimize GPU (Wang et al., 2026; Baronio et al., 2026; Zhu et al., 2026; Woo et al., 2026; Li et al., 2026; Cao et al., 2026a; li et al., 2026; Su et al., 2026; Du et al., 2026; Dai et al., 2026; Zhou et al., 2025; Chen et al., 2026; 2025) and Ascend NPU kernels (Cao et al., 2026b; Wen et al., 2026). Compilers manage registers and layouts (Tschand et al., 2025; Ran et al., 2026; Li et al., 2025a; Nichols et al., 2025; Saba et al., 2026), and resources supply examples (Wang et al., 2025), while RL with unit-test feedback improves code generation (Le et al., 2022). Rvv assembly additionally demands explicit ISA/ABI compliance at every instruction.

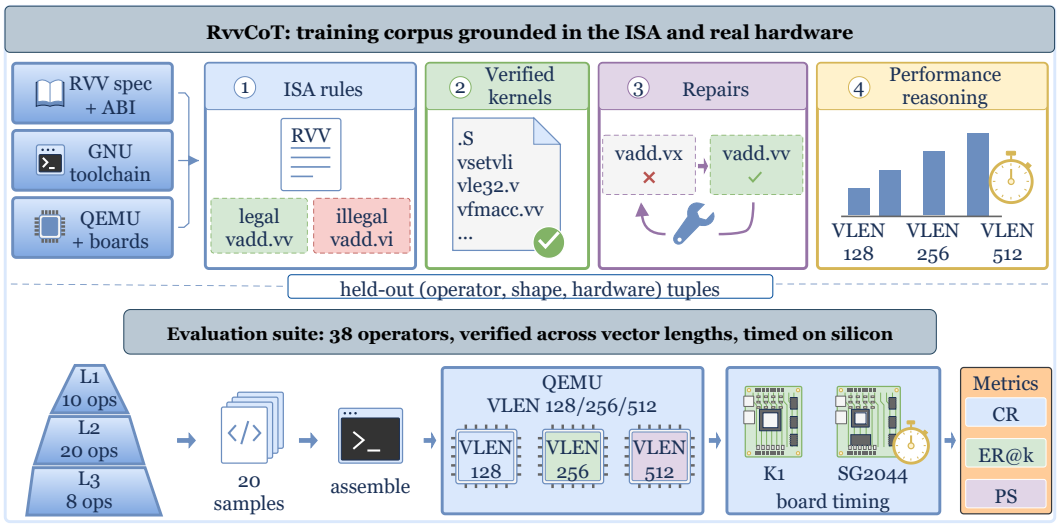


Figure 3: Overview of RvvBench: RvvCoT builds four training layers from the specification, toolchain, and boards, and a 38-operator suite checks kernels across VLEN and on boards.

3 Task: ISA-Constrained Rvv Kernel Generation

Input $x = (D_{\text{op}}, D_{\text{isa}}, D_{\text{env}})$ specifies operator, ISA, and environment requirements. Rvv assembly a must assemble/link, obey the caller's interface and ISA/ABI rules, and safely match references within operator tolerances domain-wide (Figure 2). Vector-length-agnostic loops (Perotti et al., 2022) advance by active length `vl`, preserving tail, reduction, and fused-stage semantics.

For (a, x), C_{ISA}, C_{ABI}, and C_{func} indicate ISA, ABI, and functional compliance, respectively (1: satisfied; 0: violated). $T_h(a; x)$ denotes runtime on hardware h. Since C_{func} must hold at every required `VLEN` and on every core, one kernel serves diverse hardware; we seek the fastest kernel $a^\star$:

$$a^\star \in \arg\min_a T_h(a; x) \quad \text{s.t.} \quad C_{\text{ISA}}(a, x) = C_{\text{ABI}}(a, x) = C_{\text{func}}(a, x) = 1, \tag{1}$$

$\arg\min_a$ denotes minimizers, $\in$ membership, and s.t. constraints. Correctness precedes speed and is established by compilation, execution tests across vector lengths, and timing. Hardware thus enters as a cross-length correctness constraint and an objective timed on physical boards.

4 RvvBench

RvvBench combines RvvCoT training data with held-out evaluation of ISA-constrained Rvv generation, measuring compilation, correctness, and real-hardware efficiency (Figure 3).

4.1 RvvCoT: Training Corpus Construction

RvvCoT synthesizes/distills four layers: ISA/ABI rules, verified kernel reasoning, repairs of SFT training failures, and correct kernels' hardware-performance reasoning. Samples pair x with $y = (z, a)$: reasoning z and assembly a. Compilation, reference, ISA, and deduplication checks filter targets; operator–shape–hardware splits isolate tests from synthesis, repair, and training feedback.

4.2 Evaluation Suite and Protocol

RvvBench evaluates 38 operators across three increasingly complex levels: 10 elementwise/streaming operators, 20 structured operators with reuse or reductions, and eight fused/composite operators. These levels differ from RvvCoT's layers. Each operator contributes multiple shape–hardware tasks with diagnostic inputs for tails and reductions and required `VLEN` settings, so each task checks tail handling, reduction accuracy, and portability across vector lengths.

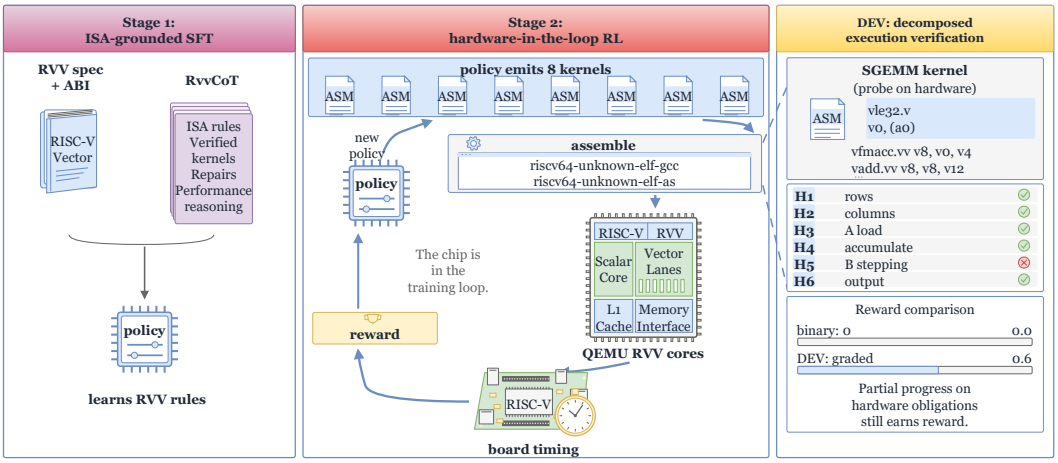


Figure 4: Overview of RvvCoder: ISA-grounded SFT learns Rvv rules, then hardware-in-the-loop RL runs eight kernels per prompt in QEMU and on boards, where DEV grades progress.

For task t, n_t, m_t, c_t count generated, compiled/linked, and correct candidates. A correct candidate must pass every diagnostic input under every required `VLEN` setting. Compilation rate $\mathrm{CR} = \sum_t m_t / \sum_t n_t$ pools tasks; execution rate $\mathrm{ER}_t@k$ uses k candidates (Chen et al., 2021):

$$\mathrm{ER}_t@k = 1 - \binom{n_t - c_t}{k} \Big/ \binom{n_t}{k}, \qquad 1 \le k \le n_t, \tag{2}$$

ER is the probability of a correct candidate among k; binomial coefficients count size-k subsets, so their ratio is the all-failing fraction. We average tasks per operator, then operators uniformly.

For task t on hardware h, $\mathcal{B}_{t,h}$ holds applicable, semantically matching baselines b with runtimes $T_b(t,h)$ under identical test inputs, data types, and harnesses; $T_{\mathrm{base}}(t,h)$ is their fastest runtime. All runtimes are medians of repeated runs on physical Rvv boards rather than emulated timings:

$$T_{\mathrm{base}}(t,h) = \min_{b \in \mathcal{B}_{t,h}} T_b(t,h), \qquad \mathrm{PS}_{t,h}@k = \frac{T_{\mathrm{base}}(t,h)}{\min_{a \in \mathcal{C}^{(k)}_{t,h}} T_a(t,h)}, \tag{3}$$

$\mathcal{C}^{(k)}_{t,h}$ contains correct, stably timed candidates a among k generations, with runtimes $T_a(t,h)$. Minima select fastest members; performance score $\mathrm{PS}_{t,h}@k$ is the baseline/candidate runtime ratio. PS thus credits speed only for verified kernels, as measured against baselines on real hardware.

5 Methodology: RvvCoder

Writing Rvv kernels requires ISA knowledge and observation on vector hardware. RvvCoder first learns ISA and ABI rules from RvvCoT by supervised fine-tuning (SFT), then improves by reinforcement learning (RL) on kernels run in QEMU and on boards, graded by DEV.

5.1 Stage 1: ISA-Grounded Supervised Fine-Tuning

Starting from Qwen3-14B (Yang et al., 2025), policy π_θ with trainable parameters θ maps task context x to response $y = (z, a)$: reasoning z followed by assembly a. As x specifies operator, ISA, and hardware, every token is conditioned on the target vector unit. The response probability is

$$\pi_\theta(y \mid x) = \prod_{t=1}^{|y|} \pi_\theta(y_t \mid x, y_{<t}), \tag{4}$$

$|y|$ counts response tokens; t indexes token y_t, and $y_{<t}$ is its prefix. The product factorizes generation autoregressively, allowing assembly tokens to condition on the preceding reasoning for each task. The reasoning thus fixes `vl` handling, `LMUL`, and registers before committing to assembly.

Let $\mathcal{D}_\ell$ distribute context–response pairs in layer $\ell \in \{1, 2, 3, 4\}$: ISA rules, verified kernels, repairs, and performance reasoning, respectively. Stage 1 samples from their joint supervised mixture $\mathcal{D}_{\text{sft}}$:

$$\mathcal{D}_{\text{sft}} = \sum_{\ell=1}^{4} \alpha_\ell \mathcal{D}_\ell, \qquad \alpha_\ell \geq 0, \qquad \sum_{\ell=1}^{4} \alpha_\ell = 1, \tag{5}$$

Here α_ℓ is layer ℓ's sampling probability, with nonnegative weights summing to one. The mixture combines ISA legality, kernel composition, error correction, and performance timed on boards.

SFT minimizes negative log-likelihood $\mathcal{L}_{\text{SFT}}$, increasing the probability of verified responses. Here $\mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{sft}}}$ averages over sampled context–response pairs, and $\log$ denotes the natural logarithm:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{sft}}} \sum_{t=1}^{|y|} \log \pi_\theta(y_t \mid x, y_{<t}). \tag{6}$$

The sum covers all target tokens, teaching reasoning and assembly jointly. Traces connect ISA constraints, calling conventions, vector-length-agnostic loops, and hardware performance choices.

Error-aware SFT adds verified teacher repairs of training failures, retaining faulty code as context and excluding test tuples. Its checkpoint initializes RL and the fixed reference policy π_{ref}. Execution feedback must distinguish newly generated candidates by their actual outcomes during training.

5.2 Stage 2: Hardware-in-the-Loop Reinforcement Learning with DEV

Group Relative Policy Optimization (GRPO) (Shao et al., 2024) learns from execution. For context x, sample $G = 8$ responses y_i with verifier rewards $r_i = r(y_i, x)$; indices i, j range from 1 to G:

$$\bar{r} = \frac{1}{G}\sum_{j=1}^{G} r_j, \qquad \sigma_r = \sqrt{\frac{1}{G}\sum_{j=1}^{G}(r_j - \bar{r})^2}, \qquad A_i = \frac{r_i - \bar{r}}{\sigma_r + \epsilon}. \tag{7}$$

$\bar{r}$ and σ_r are reward mean and population standard deviation; $\epsilon > 0$ prevents zero division. Advantage A_i favors higher-reward candidates; uniform rewards provide no within-group preference.

DEV distinguishes partial progress when whole-kernel rewards tie. For operator o, K_o diagnostics indexed by k return $h_k(y, x) \in \{0, 1\}$ on post-processed assembly; w_k weights each semantic test:

$$p_{\text{DEV}}(y, x) = \sum_{k=1}^{K_o} w_k h_k(y, x), \qquad w_k \geq 0, \qquad \sum_{k=1}^{K_o} w_k = 1. \tag{8}$$

Diagnostics return 1 for pass, 0 otherwise; $p_{\text{DEV}} \in [0, 1]$ measures weighted progress. Tests probe hardware obligations: strip-mined loop coverage, pointer stepping, reductions, and fusion.

Binary $c(y, x)$ marks extraction/compilation/linking success. Performance reward r_{perf} is zero without full correctness and valid board timings. Uncapped reward $\widetilde{r}_{\text{DEV}}$ yields final reward r_{DEV}:

$$\widetilde{r}_{\text{DEV}}(y, x) = \begin{cases} -0.20, & c(y, x) = 0, \\ 0.10 + 0.70 p_{\text{DEV}}(y, x) + r_{\text{perf}}(y, x), & c(y, x) = 1, \end{cases} \tag{9}$$

Failure earns -0.20; compilation earns 0.10, and progress has weight 0.70. GRPO uses $r = r_{\text{DEV}}$, obtained by capping $\widetilde{r}_{\text{DEV}}$ at 1.0 only after compilation and all DEV subtests pass in QEMU.

For fixed context x, write $p_\theta(y) = \pi_\theta(y \mid x)$ and $p_{\text{ref}}(y) = \pi_{\text{ref}}(y \mid x)$. The KL penalty $\mathcal{R}_{\text{KL}}$ limits drift from the fixed SFT reference distribution, retaining domain knowledge during RL updates:

$$\mathcal{R}_{\text{KL}}(\theta; x) = \beta D_{\text{KL}}(p_\theta \| p_{\text{ref}}) = \beta \sum_y p_\theta(y) \log \frac{p_\theta(y)}{p_{\text{ref}}(y)}. \tag{10}$$

$\beta \geq 0$ controls penalty strength; D_{KL} is Kullback–Leibler divergence from the current policy to its reference. The sum covers complete responses, including termination; $\log$ is natural logarithm.

DEV rewards SGEMM row coverage despite wrong reductions, guiding exploration beyond binary success. Full-harness QEMU checks certify functionality, and physical boards measure speed. Validation selects checkpoints; final tests remain excluded from training and selection.

Table 1: RvvBench (%, $n = 20$). CR: compilation; ER@k: pass@k correctness; Avg.: operator-weighted ER@10. Bold/underline: best LLM/frontier; parentheses: gap vs. GPT-5.5 (pp).

Model	**Level 1** (10 ops)			**Level 2** (20 ops)			**Level 3** (8 ops)			**Avg.**
	CR	*ER@1*	*ER@10*	*CR*	*ER@1*	*ER@10*	*CR*	*ER@1*	*ER@10*	*ER@10*
Qwen3-14B	34	2	8 (-19)	11	0	1 (-9)	4	0	0 (-3)	3 (-10)
Qwen3-Coder	54	8	18 (-9)	23	1	5 (-5)	7	0	1 (-2)	8 (-5)
DeepSeek-V4-Pro	59	9	22 (-5)	26	2	7 (-3)	10	0	2 (-1)	10 (-3)
GLM-5	42	5	13 (-14)	17	0	3 (-7)	5	0	0 (-3)	5 (-8)
Kimi-k2.5	47	6	15 (-12)	18	1	4 (-6)	6	0	0 (-3)	6 (-7)
MiniMax-M2	39	4	10 (-17)	15	0	2 (-8)	4	0	0 (-3)	4 (-9)
GPT-5.5	66	12	27 (+0)	34	3	10 (+0)	13	1	3 (+0)	13 (+0)
Gemini-3.1-Pro	62	10	24 (-3)	31	2	8 (-2)	12	0	2 (-1)	11 (-2)
Claude-Sonnet-4.6	64	11	26 (-1)	32	3	9 (-1)	11	1	3 (+0)	12 (-1)
RvvCoder-SFT	82 (+16)	44 (+32)	66 (+39)	51 (+17)	13 (+10)	28 (+18)	24 (+11)	4 (+3)	10 (+7)	34 (+21)
RvvCoder	**96** (+30)	**66** (+54)	**86** (+59)	**77** (+43)	**31** (+28)	**57** (+47)	**52** (+39)	**14** (+13)	**31** (+28)	**59** (+46)

6 Experiments and Analysis

This section presents the experimental setup, main results, and analysis. We answer the following research questions (RQs): **RQ1:** Does RvvCoder outperform frontier LLMs on RvvBench? **RQ2:** Does each core component work, and how do variants compare? **RQ3–5:** How do the hardware constraints of the task, RvvBench, and hardware-in-the-loop RL with DEV contribute? **RQ6–7:** How fast are verified kernels on real hardware, and how do they appear in concrete kernels?

6.1 Experimental Setup

Protocol and metrics. We evaluate all methods on RvvBench's 38 operators across three levels, reporting compilation rate (CR), execution correctness (ER@k), and performance score (PS). QEMU verifies functionality under multiple `VLEN` settings; performance is measured on real Rvv hardware.

Baselines. We compare with frontier LLMs, GCC `-O3`, Rvv-backend TVM (Chen et al., 2018), and expert libraries OpenBLAS (Xianyi et al., 2012) and llama.cpp (Liu et al., 2025). GCC covers every operator, the others apply where supported, and speedup uses the strongest applicable baseline.

Generation and verification. All LLMs sample $n = 20$ candidates per task under the same decoding budget, a fixed temperature schedule, and `top_p`$= 0.95$, reporting ER@k at $k \in \{1, 10\}$. Kernels are compiled statically and verified against references under VLEN $\in \{128, 256, 512\}$.

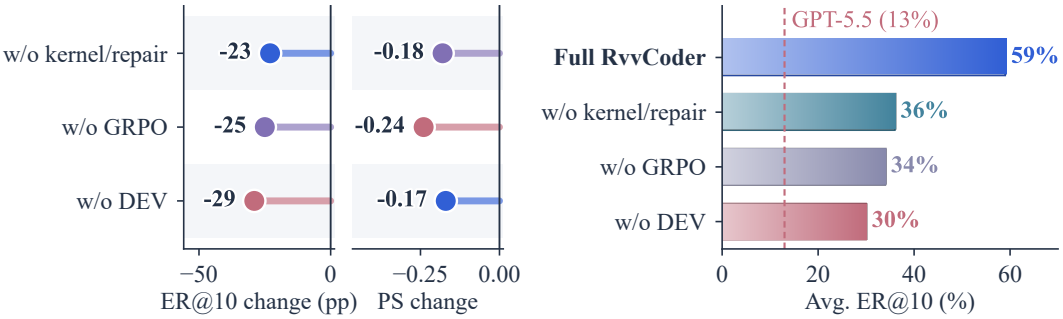
Hardware and training. Kernels are timed on SpacemiT K1/X60 and Sophgo SG2044/C920 boards by the median runtime of repeated runs. RvvCoder first fine-tunes Qwen3-14B on RvvCoT and then applies GRPO with execution feedback from QEMU or verified hardware timing.

6.2 Main Results (RQ1)

As shown in Table 1, where parentheses give each gap to GPT-5.5, RvvCoder consistently outperforms all baselines on every level and metric. In addition, we have two key observations.

Rvv hardware constraints separate compilation from correctness. Frontier models compile far more often than they compute correctly: GPT-5.5, the strongest frontier LLM, attains 66% CR but only 12% ER@1 on Level 1, and its ER@10 falls to 10% and 3% on Levels 2 and 3. RvvCoder leads it in each of the ten columns, by 30, 43, and 39 points of CR and by 59, 47, and 28 points of ER@10 on Levels 1–3, reaching 59% Avg. ER@10, a 46-point gain and $4.5\times$ its 13%.

ISA supervision and hardware feedback provide successive gains. On the same Qwen3-14B backbone, RvvCoT SFT alone already leads GPT-5.5 in every column, by 21 points of Avg. ER@10, and the second stage more than doubles this lead to 46 points. Execution in QEMU and on physical boards helps most on the harder levels: on Levels 2 and 3, the ER@10 gap grows from 18 and 7 points to 47 and 28 points, and the ER@1 gap from 10 and 3 points to 28 and 13 points.



(a) Changes relative to full RvvCoder

(b) Avg. ER@10 of each variant

Figure 5: Ablation results. (a) Changes without each component. (b) Avg. ER@10 of each variant.

6.3 Ablation Study and Comparative Analysis (RQ2)

As shown in Figure 5, we conduct an ablation study and comparative analysis on RvvCoder.

Ablation Study. We remove the kernel/repair layers of RvvCoT, the GRPO stage, and the DEV reward in turn. As shown in Figure 5(a), removing any component lowers both correctness and on-board efficiency: Avg. ER@10 falls by 23, 25, and 29 points, and PS by 0.18, 0.24, and 0.17.

Comparison across Variants. As shown in Figure 5(b), GRPO with a whole-kernel reward reaches 30% Avg. ER@10, whereas GRPO with DEV reaches 59%, so graded DEV feedback on executions makes the second stage effective. Every ablated variant still exceeds GPT-5.5 by at least 17 points.

6.4 Analysis of ISA-Constrained Task Difficulty (RQ3)

As shown in Figure 6, we analyze how compilation relates to correctness for all models.

Compilation does not imply correctness. Every frontier point lies far below the CR = ER diagonal: GPT-5.5 reaches 66% CR but 12% ER@1 on Level 1, and no frontier LLM exceeds 1% ER@1 on Level 3. RvvCoder instead occupies the upper-right region at every level, with CR/ER@1 of 96/66, 77/31, and 52/14. For the untuned Qwen3-14B, 81% of candidates fail before execution and only 2% pass (Table 15), so Rvv ISA/ABI rules and functional semantics are both active bottlenecks.

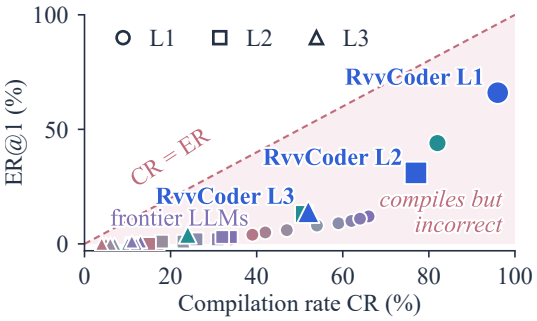


Figure 6: CR versus ER@1 for all 11 models.

6.5 Analysis of RvvBench Supervision and Evaluation (RQ4)

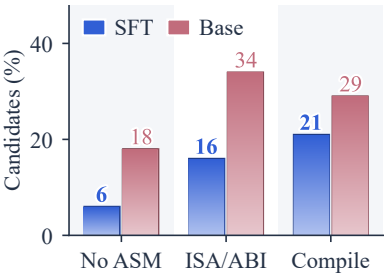
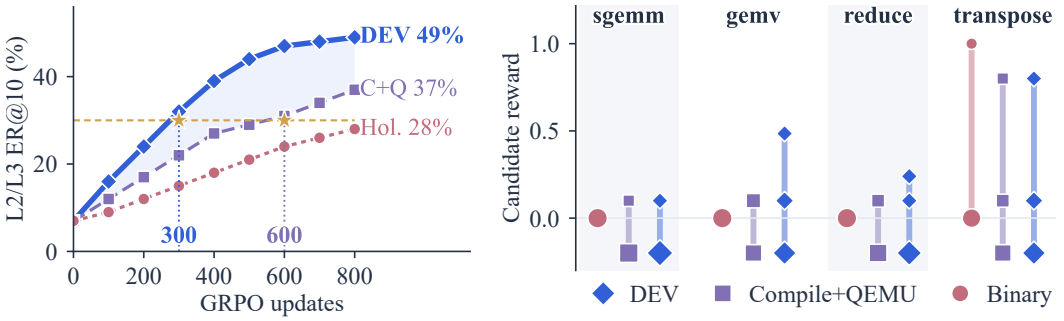


Figure 7: Candidate outcomes.

As shown in Figure 7, we analyze how RvvCoT supervision reshapes the candidate outcomes of the same backbone.

RvvCoT supervision improves every level and metric. On the same backbone, SFT on RvvCoT raises all nine level–metric scores in Table 1 and lifts Avg. ER@10 from 3% to 34%. It also cuts constraint violations: no-assembly outputs fall from 18% to 6%, ISA/ABI violations from 34% to 16%, and compilation failures from 29% to 21%, so pre-execution failures drop from 81% to 43% and passing ones grow from 2% to 16% (Table 15). Thus, RvvCoT supplies the Rvv hardware rules and kernel knowledge the backbone lacks.



(a) Reward-design learning curves

(b) Rewards within GRPO-sized groups ($G = 8$)

Figure 8: The results of reward design. (a) L2/L3 ER@10 of three rewards (C+Q: Compile+QEMU; Hol.: holistic; gold: 30% target). (b) Rewards of eight candidates for each Level-2 operator.

6.6 Analysis of Execution-Grounded RL with DEV (RQ5)

As shown in Figure 8, we analyze how DEV turns executed kernels into learning signals.

DEV learns faster and reaches higher correctness. As shown in Figure 8(a), DEV reaches the 30% target after 300 updates, twice as fast as Compile+QEMU, while the holistic reward stays below it within 800 updates. DEV leads at every checkpoint and ends at 49%, versus 37% and 28%.

DEV keeps GRPO groups informative. As shown in Figure 8(b), a binary reward gives identical rewards in three of four groups, which yields zero advantage, whereas DEV separates candidates in all four; a GEMV kernel passing two of four sub-properties earns 0.485, versus 0.10 for none.

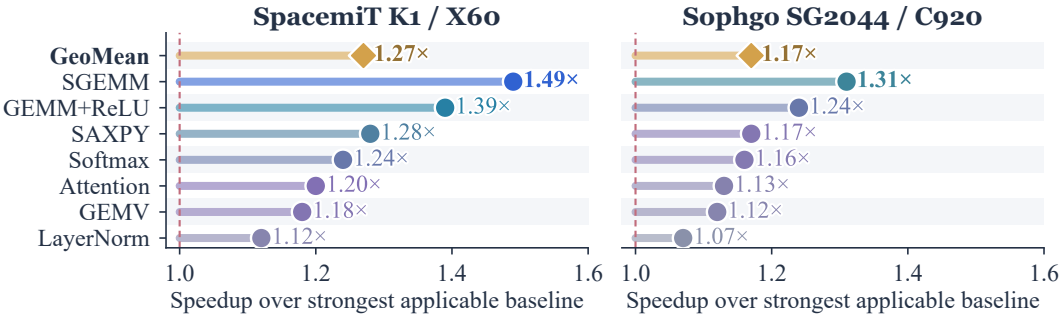


Figure 9: Real-hardware speedups over the strongest applicable baseline; dashed line: parity.

6.7 Real-Hardware Efficiency (RQ6)

As shown in Figure 9, we measure the speedups of verified kernels on two Rvv boards.

Verified kernels are faster on both boards. Only verified kernels are timed. On SpacemiT K1/X60, all seven operators beat the strongest applicable baseline, with a 1.27× geometric mean; SGEMM gains the most at 1.49×, or 33% less runtime, followed by GEMM+ReLU at 1.39×.

Gains transfer across microarchitectures. On Sophgo SG2044/C920, the geometric mean is 1.17× and SGEMM again leads at 1.31×. The seven operators keep the same ordering on both boards (Spearman $\rho = 1.00$), so the gains stem from kernel design rather than a single core.

6.8 Case Study (RQ7)

As shown in Figure 10, we analyze four representative failures of generated Rvv kernels.

RvvCoT resolves ISA-legality failures. Case 1 opens with the ARM directive `.syntax unified`, which the RISC-V assembler rejects at line 1, and Case 2 misorders the vtype fields of

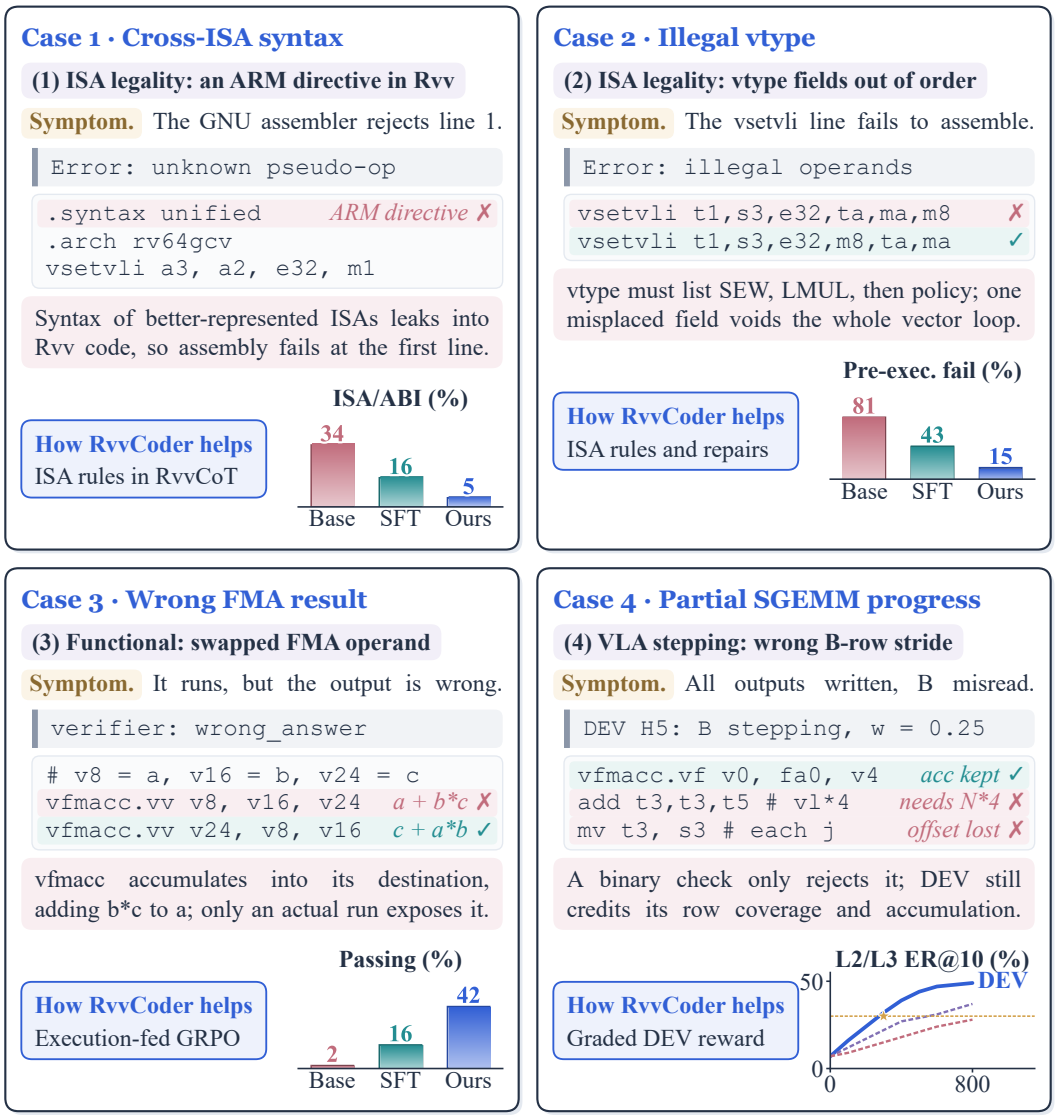


Figure 10: Case study on four failures of generated kernels, each tied to a task constraint and to the RvvCoder component that resolves it, with insets comparing Base, SFT, and Ours.

`vsetvli`. Both reflect scarce Rvv supervision, and the ISA rules and verified repairs of RvvCoT cut ISA/ABI violations from 34% to 5% and pre-execution failures from 81% to 15%.

Execution feedback resolves semantic failures. Case 3 links yet computes $a + b \cdot c$ instead of $a \cdot b + c$, which only execution exposes; execution-grounded GRPO raises passing candidates from 2% to 42%. Case 4 advances B by `vl` $\times$ 4 bytes instead of the row pitch, yet DEV still credits its coverage and accumulation. The final SGEMM kernel runs up to $1.49\times$ faster than baselines.

7 Conclusion

This work tackles generating correct and fast assembly for vector-length-agnostic RISC-V hardware. We formalize ISA-constrained Rvv kernel generation, build RvvBench, whose 38-operator suite is verified across vector lengths and timed on real boards, and present RvvCoder, which learns Rvv rules and improves by executing its kernels with DEV feedback. RvvCoder reaches 59% Avg. ER@10, versus 13% for GPT-5.5, and runs up to $1.49\times$ faster than the strongest applicable baselines on two boards. Future work will extend RvvCoder to more ISAs, operators, and hardware platforms.

AI USE STATEMENT

We used generative AI tools only for language polishing and formatting assistance. The LLMs that form part of the method or serve as baselines are described in the main text. The authors reviewed all AI-assisted edits and take full responsibility for the complete content of this paper.

ETHICS STATEMENT

This work involves no human subjects or personal data. RvvCoT and RvvBench are built from public ISA specifications, open-source toolchains, and synthesized kernels, used under their licenses. All generated kernels are verified in QEMU or an isolated harness before any hardware execution.

REPRODUCIBILITY STATEMENT

Sections 3–5 define the task, RvvBench, and the training method, and the appendix details the corpus construction, evaluation protocol, baselines, hardware, hyperparameters, and DEV harnesses. The code, benchmark tasks, and training configurations will be released upon publication.

REFERENCES

Imad Al Assir, Mohamad El Iskandarani, Hadi Rayan Al Sandid, and Mazen A. R. Saghir. Arrow: A risc-v vector accelerator for machine learning inference, 2021. URL https://arxiv.org/abs/2107.07169.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 83418–83452, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/86ba836d4c5dd859d795a172911745e2-Paper-Conference.pdf.

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

Xinzi Cao, Jianyang Zhai, Pengfei Li, Zhiheng Hu, Cen Yan, Mubingxu, Guanghuan Fang, Bin She, Jiayu Li, Yihan Su, Dongyang Tao, Feidiao Yang, Chang-Dong Wang, Yutong Lu, Weicheng Xue, Bin Zhou, and Yonghong Tian. AscendKernelGen: LLM-driven kernel generation for NPUs. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 30693–30718, San Diego, California, United States, July 2026b. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1533. URL https://aclanthology.org/2026.findings-acl.1533/.

Jinwu Chen, Qidie Wu, Bin Li, Lin Ma, Xin Si, Yang Hu, Shouyi Yin, and Jun Yang. cupilot: A strategy-coordinated multi-agent framework for cuda kernel evolution, 2025. URL https://arxiv.org/abs/2512.16465.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Terry Chen, Zhifan Ye, Bing Xu, Zihao Ye, Timmy Liu, Ali Hassani, Tianqi Chen, Andrew Kerr, Haicheng Wu, Yang Xu, Yu-Jung Chen, Hanfeng Chen, Aditya Kane, Ronny Krashinsky, Ming-Yu Liu, Vinod Grover, Luis Ceze, Roger Bringmann, John Tran, Wei Liu, Fung Xie, Michael

Lightstone, and Humphrey Shi. Avo: Agentic variation operators for autonomous evolutionary search, 2026. URL https://arxiv.org/abs/2603.24517.

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 *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pp. 579–594, USA, 2018. USENIX Association. ISBN 9781931971478.

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, 2026. URL https://arxiv.org/abs/2602.24286.

He Du, Qiming Ge, Jiakai Hu, Aijun Yang, Zheng Cai, Zixian Huang, Sheng Yuan, QinXiu Cheng, XIE Xinchen, Yicheng Chen, Yining Li, Jiaxing xie, Huanan Dong, Yaguang Wu, Xiangjun Huang, jian yang, Hui Wang, Bowen Zhou, Bowen Li, Qipeng Guo, and Kai Chen. Kernelsmith: A unified recipe for evolutionary kernel optimization. In *Deep Learning for Code: Towards Human-Centered Coding Agents*, 2026. URL https://openreview.net/forum?id=IMV70cIb8K.

Yuxuan Han, Meng-Hao Guo, Zhengning Liu, Wenguang Chen, and Shi-Min Hu. Making llms optimize multi-scenario cuda kernels like experts, 2026. URL https://arxiv.org/abs/2603.07169.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards Robust Agentic CUDA Kernel Benchmarking, Verification, and Optimization, 2025. URL https://arxiv.org/abs/2509.14279.

C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, 2004. doi: 10.1109/CGO.2004.1281665.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 21314–21328. Curran Associates, Inc., 2022. doi: 10.52202/068431-1549. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/8636419dea1aa9fbd25fc4248e702da4-Paper-Conference.pdf.

Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, Pi-Feng Chiu, Rimas Avizienis, Brian Richards, Jonathan Bachrach, David Patterson, Elad Alon, Bora Nikolic, and Krste Asanovic. An agile approach to building risc-v microprocessors. *IEEE Micro*, 36(2):8–20, 2016. doi: 10.1109/MM.2016.11.

Haonan Li, Keyu Man, Partha Kanuparthy, Hanning Chen, Wei Sun, Sreen Tallam, Chenguang Zhu, Kevin Zhu, and Zhiyun Qian. Tritonforge: Profiling-guided framework for automated triton kernel optimization, 2025a. URL https://arxiv.org/abs/2512.09196.

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 Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23053–23066, Vienna, Austria, July 2025b. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.1183. URL https://aclanthology.org/2025.findings-acl.1183/.

Ruo-Shi Li, Ping Peng, Zhi-Yuan Shao, Hai Jin, and Ran Zheng. Evaluating risc-v vector instruction set architecture extension with computer vision workloads. *Journal of Computer Science and Technology*, 38(4):807–820, 2023.

ShangZhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, jianling li, YONGGANG HU, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. AUTOTRITON: Automatic triton programming with reinforcement learning in LLMs, 2026. URL https://openreview.net/forum?id=8m6yY1ULKh.

xiaoya li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 123922–123957, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/c94bbbef466ab1b2cfa100e41413b3a8-Paper-Conference.pdf.

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

Zhilong Liu, Long Peng, Wenzhu Wang, Ke Li, Binrui Zeng, Jie Yu, and Xiaodong Liu. Accelerating llm inference on risc-v edge devices via vector extension optimization. In De-Shuang Huang, Chuanlei Zhang, Qinhu Zhang, and Yijie Pan (eds.), *Advanced Intelligent Computing Technology and Applications*, pp. 515–526, Singapore, 2025. Springer Nature Singapore. ISBN 978-981-96-9869-1.

Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

Daniel Nichols, Konstantinos Parasyris, Charles Jekel, Abhinav Bhatele, and Harshitha Menon. Integrating performance tools in model reasoning for gpu kernel optimization, 2025. URL https://arxiv.org/abs/2510.17158.

D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pp. 11 pp.–294, 2006. doi: 10.1109/CGO.2006.25.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Matteo Perotti, Matheus Cavalcante, Nils Wistoff, Renzo Andri, Lukas Cavigelli, and Luca Benini. A "new ara" for vector computing: An open source highly efficient risc-v v 1.0 vector processor design. In *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 43–51, 2022. doi: 10.1109/ASAP54787.2022.00017.

Anna Pirova, Anastasia Vodeneeva, Konstantin Kovalev, Alexander Ustinov, Evgeny Kozinov, Alexey Liniov, Valentin Volokitin, and Iosif Meyerov. Performance optimization of blas algorithms with band matrices for risc-v processors. *Future Generation Computer Systems*, 174:107936, 2026. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2025.107936. URL https://www.sciencedirect.com/science/article/pii/S0167739X25002316.

Dezhi Ran, Shuxiao Xie, Mingfang Ji, Anmin Liu, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Hao Yu, Linyi Li, Yitao Hu, Wei Yang, and Tao Xie. Kernelband: Steering LLM-based kernel optimization via hardware-aware multi-armed bandits. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=d8e2jBfMpE.

Tara Saba, Zhiyang Chen, Jikai Jason Li, Anne Ouyang, Xujie Si, and Fan Long. Cutegen: An llm-based agentic framework for generation and optimization of high-performance gpu kernels using cute, 2026. URL https://arxiv.org/abs/2604.01489.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Songqiao Su, Xiaoya Li, Albert Wang, Guoyin Wang, Jiwei Li, and Chris Shum. Cuda-l2: Surpassing cublas performance for matrix multiplication through reinforcement learning, 2026. URL https://arxiv.org/abs/2512.02551.

Arya Tschand, Kesavan Ramakrishnan, Muhammad A. Awad, Ryan Swann, Jeffrey Jian Ma, Keith Lowery, and Vijay Janapa Reddi. Swizzleperf: Hardware-aware LLMs for GPU kernel performance optimization. In *Machine Learning for Systems 2025*, 2025. URL https://openreview.net/forum?id=a5aJi9OAr0.

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

Yaoyu Wang, Hankun Dai, Zhidong Yang, Junmin Xiao, and Guangming Tan. Mastering Sparse CUDA Generation through Pretrained Models and Deep Reinforcement Learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=VdLEaGPYWT.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. Multikernelbench: A multi-platform benchmark for kernel generation, 2025. URL https://arxiv.org/abs/2507.17773.

Zhongzhen Wen, Shudi Shao, Zhong Li, Yu Ge, Tongtong Xu, Yuanyi Lin, and Tian Zhang. Ascendcraft: Automatic ascend npu kernel generation via dsl-guided transcompilation, 2026. URL https://arxiv.org/abs/2601.22760.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. TritonRL: Training LLMs to think and code triton without cheating. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=LIlJXdstH2.

Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pp. 684–691, 2012. doi: 10.1109/ICPADS.2012.97.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Qirui Zhou, Shaohui Peng, Weiqiang Xiong, Haixin Chen, Yuanbo Wen, Haochen Li, Ling Li, Qi Guo, Yongwei Zhao, Ke Gao, Ruizhi Chen, Yanjun Wu, Zhao Chen, and Yunji Chen. QiMeng-attention: SOTA attention operator is generated by SOTA attention algorithm. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 8491–8505, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.446. URL https://aclanthology.org/2025.findings-acl.446/.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, et al. Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29168–29176, 2026.

A RVV TASK CONSTRAINTS

This appendix expands the task definition of Section 3. The input $x = (D_{\text{op}}, D_{\text{isa}}, D_{\text{env}})$ has three parts: the operator specification D_{op} gives the operator semantics, shapes, data types, layouts, and strides; the ISA specification D_{isa} gives the target ISA and its allowed vector configurations; and the environment specification D_{env} gives the hardware, required `VLEN` settings, exported symbol, and calling convention. The output is callable assembly that assembles and links with the caller, receives arguments through the interface, and writes its results to the specified memory.

Hardware `VLEN` is the number of bits in one vector register and is fixed for a core, whereas the active length `vl` is set at run time. The instruction `vsetvli` writes `vl` and the vector type `vtype`, which holds the element width `SEW`, the register-group multiplier `LMUL`, and the tail and mask policies. A vector-length-agnostic loop requests the remaining element count, advances by the returned `vl`, and therefore runs unchanged on cores with different `VLEN`.

Rvv (Perotti et al., 2022) encodes `LMUL` values $\{1/8, 1/4, 1/2, 1, 2, 4, 8\}$; usable `SEW`/`LMUL` combinations depend on the element widths and configuration rules of the target. Register groups require aligned starting registers, with instruction-dependent group sizes and overlap restrictions, and a larger group leaves fewer independent live vector groups. Table 2 lists the five constraint categories of the task with their representative failure modes.

Table 2: Rvv constraints and representative failure modes for generated assembly.

Constraint	**Required behavior**	**Representative failure**
`vsetvli` and strip-mining	Use the actual `vl` for progress, update byte addresses consistently, and handle tails without invalid accesses.	Skipped or repeated elements, out-of-bounds accesses, or errors at a different `VLEN`.
`LMUL` and register groups	Respect supported vector configurations, group alignment, operand overlap rules, and the lifetimes of intermediate values.	Assembly rejection, illegal-instruction exceptions, or corrupted live values.
Register classes	Use integer, floating-point, and vector operands in their required register files, with valid transfer instructions.	Rejected operands or incorrect values from confused register classes.
Instruction and addressing legality	Use supported mnemonics, operand forms, and memory-addressing modes for the target ISA.	Nonexistent instructions, cross-ISA syntax, or unsupported operand combinations.
ABI and calling interface	Export the required symbol, pass arguments and results correctly, and preserve registers and stack state as required by the selected convention.	Link failures, incorrect arguments or results, or corruption of the caller's state.

Satisfying these local constraints is necessary but not sufficient for whole-kernel semantics. Correct initialization, loop coverage, pointer stepping, reduction completion, and producer–consumer dependencies must jointly realize the operator. For example, advancing a contiguous float32 pointer by `vl` elements requires a byte offset of $4\,\texttt{vl}$, and strided accesses also scale it by the stride.

B RVVCOT CONSTRUCTION DETAILS

This appendix expands Section 4.1. Each training unit is an operator–shape–hardware tuple (o, s, h): the operator o fixes the semantics and exported symbol, the shape s fixes tensor dimensions, strides, and tail cases, and the hardware descriptor h fixes `SEW`, candidate `LMUL` values, the calling convention, and the target `VLEN`. Each sample pairs a context x with a response $y = (z, a)$, where z is an explanation and a is Rvv assembly. Table 3 lists the construction prompts of the four layers.

Layer 1: ISA Constraint CoT. We parse the Rvv specification and ABI rules into instruction- and constraint-level topics, including `vsetvli` semantics, `SEW`/`LMUL` compatibility, legal register groups, integer versus floating-point register classes, memory access forms, and calling conventions.

These topics become question–answer examples, and frequent LLM hallucinations become correction triples of the form *wrong, correct, reason*, which expose nonexistent mnemonics, fixed-width SIMD assumptions, invalid addressing modes, and cross-ISA syntax.

Layer 2: Kernel Synthesis and Reasoning Distillation. For each tuple, parameterized assembly skeletons enumerate `LMUL` settings, strip-mining strategies, register-allocation patterns, and memory layouts. A candidate is retained only after it assembles, matches reference outputs under several QEMU `VLEN` settings, and passes static ISA checks. A teacher model then reconstructs the rationale behind the key implementation choices, and we keep only explanations that are consistent with the verified assembly and the Rvv constraints.

Layer 3: Self-Play Error Correction. The initial SFT policy samples kernels on training tuples, and the verifier sorts its failures into Rvv-specific categories such as illegal register groups, ABI violations, wrong outputs, and timeouts. Conditioned on the failure category and the verifier message, a teacher model writes a diagnosis and a repaired implementation. Only repairs that pass the verifier become targets, and the failed program is kept as context rather than used as an answer.

Layer 4: Performance-Reasoning CoT. For functionally correct kernels, we time legal alternative configurations on real Rvv hardware with repeated runs and ask the teacher to explain the measured differences. Typical explanations relate `LMUL` to register pressure, `vsetvli` placement to loop overhead, and accumulator reuse to memory traffic. Explanations are conditioned on measured timings; QEMU supplies correctness only and is never used as a timing source.

Table 3: Prompt templates used to construct RvvCoT.

Layer	Template
L1 constraint QA	`You are an Rvv ISA tutor. Given the Rvv rule [rule_name] and target configuration [SEW, LMUL, VLEN], answer the question and explain the constraint. Focus on legal operands, register classes, vsetvli behavior, ABI conventions, and why common alternatives are invalid. Do not write a full benchmark kernel.`
L1 correction triple	`The following Rvv assembly fragment is wrong: [bad_fragment]. Identify the error category, provide the corrected fragment, and explain the minimal ISA rule that justifies the correction. Use the format: Error / Fix / Reason.`
L2 kernel synthesis	`Generate RV64GCV assembly for operator [operator] with ABI [signature]. Shapes: [shape]. Semantics: [reference_semantics]. Use vector-length-agnostic strip-mining, legal Rvv instructions, correct scalar and vector register classes, and export exactly .globl [symbol]. Return a short planning rationale followed by the assembly.`
L3 error correction	`A generated Rvv kernel for [operator] failed verification. Failure: [failure_category]. Compiler/QEMU trace: [trace]. Explain the likely root cause, identify the violated Rvv or ABI rule, and produce a corrected implementation that preserves the required symbol [symbol].`
L4 performance reasoning	`For operator [operator] on hardware [hardware] with VLEN=[vlen], compare candidate configurations [configs] and measured times [measurements]. Explain which LMUL, unroll, strip-mining, memory layout, and reduction schedule is preferred, and derive a concise implementation guideline.`

Quality control. Executable targets must assemble with the RISC-V GNU toolchain, pass reference checks under multiple QEMU `VLEN` settings, and pass a static checker. The checker rejects nonexistent mnemonics, operands in the wrong register class, nonexistent scalar registers such as `t7`, unsupported vector addressing forms, missing `.globl` symbols, and `LMUL`-misaligned destination registers. Near-duplicates are removed by comparing instruction sequences after labels, register names, stack offsets, and immediates are canonicalized, together with operator metadata; canonicalization is used only for comparison and never rewrites a target. Table 4 gives the retention rules for each source; negative examples are kept only with a known failure mode and a verified fix.

Data separation. The corpus is split by (o, s, h) tuple into training, validation, and test tuples. Test tuples are never used for synthesis, repair mining, or training feedback, which keeps all reported test results free of contamination, and the validation tuples select checkpoints. Exact matches compare the operator, shape, layout, `VLEN`, symbol, and harness signature, and near matches compare normalized semantics and instruction skeletons. For fused Level-3 operators, traces sharing the

Table 4: Retention rules for each RvvCoT source.

Source	Retain if	Discard if
ISA QA	The rule cites a valid Rvv or ABI constraint and has a unique answer	The answer depends on a benchmark solution or an unsupported ISA variant
Correction triples	The bad fragment maps to a known error category and the fix is valid	The fix changes operator semantics or introduces another illegal pattern
Kernel SFT	It compiles, passes QEMU at every `VLEN`, and passes static ISA checks	It hard-codes `VLEN`, uses illegal register groups, violates the ABI, or passes one shape by chance
Error repair	The failure is reproducible and the repaired kernel passes the verifier	The trace comes from a test tuple or the repair cannot be verified
Performance CoT	The kernel is correct, timing is stable, and the explanation matches the measured ordering	The timing is noisy, unannotated for hardware, or derived from QEMU alone

fused computation graph are removed, while their primitive components such as standalone `gemm` or `relu` may remain in training as reusable building blocks for composing new fused kernels.

C Evaluation Suite and Protocol Details

This appendix expands Section 4.2. Table 5 lists the 38 operators. Level 1 exercises local Rvv idioms such as elementwise arithmetic, streaming memory access, and simple reductions; Level 2 adds data reuse, reductions, layout-sensitive movement, and multi-loop control; and Level 3 requires consistent composition across several computation stages. Each operator contributes one or more shape–hardware tasks, so 38 counts operator kinds rather than executions.

Table 5: The 38 RvvBench operators across three difficulty levels.

Level	Operators
L1 (10)	ReLU, Add, Mul, SAXPY, ReduceSum, Elementwise, Rotate, Transpose, Gather, Scatter
L2 (20)	SGEMM, GEMM, MatMul, Conv2D, Softmax, LayerNorm, BatchNorm, MaxPool2D, MVT, GESUMMV, GEMVER, TRMM, TwoMM, FIR, Jacobi1D, Jacobi2D, Covariance, Correlation, ReduceSumND, GEMM-Int32
L3 (8)	GEMM_ReLU, SGEMM_BiasReLU, MatMul_GELU, Conv_BN_ReLU, Attention, MHA, FusedMLP, TransformerBlock

Candidate generation. All LLMs receive the same prompt family. The system prompt reads: *You are a professional RISC-V Vector Extension (RVV) kernel engineer. Generate optimized RVV assembly kernels for the specified operator and target hardware. Always specify tail and mask policies in vsetvli. Output a complete, standalone .S file with a proper .globl directive and ret.* The user prompt gives the operator semantics, ABI, shape, symbol, and hardware. Each model samples $n = 20$ candidates per task, cycling through a fixed temperature schedule between 0.2 and 1.0 with `top_p`$= 0.95$, and independent requests. Table 6 lists the queried model identifiers.

Extraction and verification. Assembly is taken from a fenced code block, or from the first `.text` or `.globl` directive, and a candidate without extractable assembly is labeled `no_asm`. The deterministic post-processor then fixes local surface errors without changing the algorithm. Each candidate is linked with the operator harness by `riscv64-linux-gnu-gcc -march=rv64gcv -mabi=lp64d -O0 -static` and executed with `qemu-riscv64` at `VLEN`$\in \{128, 256, 512\}$ under fixed timeouts. Harness inputs include tails that are not multiples of common `VLEN` values, non-square matrices, and both short and long reduction axes for softmax and normalization.

Numerical correctness. A candidate is correct only if every input case passes at every `VLEN` setting. Integer outputs must match exactly. For a floating-point output element $\hat{y}$ of the candidate and the

Table 6: Frontier LLMs and the model identifiers used for generation.

Model	Identifier
Qwen3-14B	`qwen3-14b`
Qwen3-Coder	`qwen3-coder-480b-a35b-instruct`
DeepSeek-V4-Pro	`deepseek-v4-pro`
GLM-5	`glm-5`
Kimi-k2.5	`kimi-k2.5`
MiniMax-M2	`MiniMax-M2.7`
GPT-5.5	`gpt-5.5`
Gemini-3.1-Pro	`gemini-3.1-pro-preview`
Claude-Sonnet-4.6	`claude-sonnet-4-6`

corresponding element y of the C reference, the check is

$$|\hat{y} - y| \leq \epsilon_{\text{abs}} + \epsilon_{\text{rel}} |y|, \tag{11}$$

where $|\cdot|$ is the absolute value and ϵ_{abs} and ϵ_{rel} are absolute and relative tolerances fixed per operator. RELU is checked exactly, SGEMM uses $\epsilon_{\text{abs}} = 10^{-4}$ and $\epsilon_{\text{rel}} = 10^{-5}$, and ATTENTION uses 10^{-3} and 10^{-4}. Any NaN, infinity, out-of-bounds write, ABI violation, timeout, or crash fails.

Metric aggregation. For task t, n_t counts the generated candidates, c_t the correct ones, and k is the subset size of ER@k. When $n_t - c_t < k$, every size-k subset contains a correct candidate and $\text{ER}_t@k = 1$, and for $k = 1$ the estimator reduces to $\text{ER}_t@1 = c_t/n_t$. ER is averaged over the tasks of each operator and then uniformly over operators, so operators with more shapes do not carry more weight. CR keeps failed and unextractable candidates in its denominator.

Performance coverage. Only candidates that pass the full harness are timed. A task on which no candidate is both correct and stably timed has no measured speedup and is reported as uncovered. Speedups are summarized per board by the geometric mean over covered operators.

Baselines. GCC `-O3 -march=rv64gcv` compiles the C reference of every operator and is always part of the baseline pool. TVM, OpenBLAS, and llama.cpp join the pool only when they provide a native operator, schedule, or library call with matching semantics, data type, layout, and shape contract, as summarized in Tables 7 and 8. An unavailable baseline is dropped from that pool rather than imputed, and the fastest remaining member defines the reference runtime.

Table 7: Baseline coverage by operator group; Y marks an applicable baseline.

Operator group	GCC	TVM	OpenBLAS	llama.cpp	Examples
Elementwise / streaming	Y	Y	–	–	RELU, ADD, MUL, TRANSPOSE, GATHER, SCATTER
BLAS-1/2	Y	Y	Y	–	SAXPY, MVT, GESUMMV, GEMVER
BLAS-3	Y	Y	Y	–	SGEMM, GEMM, MATMUL, TRMM, TWOMM
Tensor / stencil	Y	Y	–	–	CONV2D, MAXPOOL2D, FIR, JACOBI1D, JACOBI2D
Transformer primitives	Y	Y	–	Y	SOFTMAX, LAYERNORM, ATTENTION, MHA
Fused operators	Y	Y	–	partial	GEMM_RELU, MATMUL_GELU, CONV_BN_RELU, FUSEDMLP

Hardware and timing. Table 9 lists the execution targets. Each verified binary runs warmup iterations followed by repeated timed iterations measured with `clock_gettime`, and the median runtime is reported. All methods for the same task share the input tensors, shape parameters, harness, and timing loop, and runs are pinned to a fixed core where the platform allows it.

Failure categories. Every candidate receives one terminal category: `no_asm`, `static_invalid`, `compile_fail`, `exec_fail`, `timeout`, `wrong_answer`, or `pass`. For the candidate-

Table 8: Rules for deciding whether a baseline applies to a task.

Baseline	Applicability rule
GCC `-O3`	Always included by compiling the C reference with Rvv-enabled flags on the same inputs and harness
TVM Rvv	Included when the operator is expressible as a tensor program with matching data type, layout, boundary behavior, and fused semantics
OpenBLAS Rvv	Included only for BLAS-compatible operators whose semantics map to standard BLAS calls without changing the harness
llama.cpp Rvv	Included for transformer primitives whose exported kernel matches the data type, layout, normalization or softmax definition, and shape contract

Table 9: Execution targets; QEMU checks correctness and the physical boards provide all timings.

Target	Core	VLEN	Role	Use
SpacemiT K1	X60	256	Performance	Primary board for speedups over all applicable baselines
Sophgo SG2044	C920	128	Performance	Second board with a different core design and vector length
QEMU	emulated RV64GCV	128/256/512	Correctness	Functional verification, failure categories, and DEV harnesses

level analysis in Section 6, `static_invalid` forms the ISA/ABI class, `compile_fail` the compilation-failure class, and runtime failures the wrong-output class.

D Training and Implementation Details

This appendix expands the training stages of Section 5. Stage 1 fine-tunes Qwen3-14B on the four RvvCoT layers, and Stage 2 applies GRPO with the DEV reward from the Stage-1 checkpoint, which is also the KL reference. Table 11 lists the hyperparameters and Table 10 the training prompts.

Stage 1: domain- and error-aware SFT. The supervision target is a compact explanation followed by the assembly. The explanation states the calling convention, element type, loop structure, `vsetvli` policy, legal register groups, and tail handling before any code is emitted. Repair examples contain the failing candidate, the verifier message, the failure category, a corrected explanation, and the repaired kernel. Hard operators and frequent failure categories are up-weighted.

Stage 2: execution-grounded GRPO. For each training prompt, the policy samples $G = 8$ candidates. Each candidate is post-processed, compiled, executed by the full harness and, for structured and fused operators, by the DEV diagnostics. The group-relative objective raises the probability of candidates that satisfy more executable obligations, while the KL penalty keeps the policy close to the Stage-1 reference. Timing rewards apply only to candidates that are already fully correct.

Table 10: Prompt templates used during training, shortened for readability.

Template	Content
System	You are an expert RISC-V Rvv assembly programmer. Generate vector-length-agnostic Rvv assembly that follows the RISC-V ABI, uses legal `SEW`/`LMUL` combinations, preserves required registers, and emits only the requested kernel.
Domain SFT	Given the operator semantics, C reference behavior, tensor shapes, element type, symbol name, and target `VLEN`, first summarize the plan: arguments, loop bounds, `vsetvli` policy, register allocation, memory access pattern, and tail handling. Then output the assembly in one fenced code block.
Error-aware SFT	Given a failed kernel, compiler or execution feedback, and the reference semantics, identify the failure category, explain the minimal correction, and produce a repaired kernel without non-Rvv mnemonics, fixed-width SIMD assumptions, illegal vector addressing, or non-ABI symbols.
GRPO with DEV	Given the kernel synthesis prompt, generate one candidate kernel. Prefer simple, correct, vector-length-agnostic code, and use performance-oriented choices only when they preserve the verifier obligations.

Table 11: Training hyperparameters of RvvCoder.

Parameter	Stage 1: SFT	Stage 2: GRPO with DEV
Initialization	Qwen3-14B	Stage-1 checkpoint
Objective	Causal language-model loss on the four RvvCoT layers	Group-relative policy optimization with verifier rewards and KL to the Stage-1 policy
Learning rate	1.0–1.25×10^{-6}	2×10^{-7}
Schedule	Cosine decay with warmup	Cosine decay with warmup
Training length	2–3 epochs over the SFT mixture	500–1000 GRPO updates, evaluated every 100 updates
Batching	Effective batch size 128	$G = 8$ candidates per prompt
Sequence length	8192 tokens	8192 prompt and completion tokens
Precision	bf16 with tf32	bf16 with tf32
Adapter	LoRA rank 32 on attention and MLP projections	LoRA rank 32 initialized from the Stage-1 adapter
Sampling temperature	–	0.8 during training
KL coefficient β	–	0.03–0.10
Compute	4 NVIDIA L20X GPUs	4 NVIDIA L20X GPUs with QEMU and hardware verifier workers

Post-processing. The deterministic post-processor repairs frequent surface errors of LLM-generated Rvv assembly, summarized in Table 12. Rules cover nonexistent instructions, missing width suffixes, confusion between integer and floating-point register files, ARM branch mnemonics, indexed addressing, and incomplete `vsetvli` operands. For `LMUL`≥ 2, a remapping rule scans all vector register references, finds indices that are not multiples of `LMUL`, maps them to the nearest valid aligned groups, and rewrites every reference consistently. Every processed program is compiled and verified again, and both raw and processed outputs are logged.

Table 12: Frequent LLM errors in Rvv assembly and the corresponding post-processing rules.

#	LLM output	Correction	Error type
1	`vfmv.vi vd, imm`	`vmv.v.i vd, imm`	Nonexistent instruction
2	`vle.v / vse.v / vlw.v`	`vle32.v / vse32.v`	Missing width suffix
3	`vfmv.v.f vd, zero`	Move zero to `ft0`, then broadcast	Register file confusion
4	`vfmv.vf vd, zero, vs`	`vmv.v.i vd, 0`	Extra operand variant
5	`vfmv.v.x / vfmv.s.x`	`vmv.s.x`	Float prefix, integer move
6	`vfmv.x.s`	`vmv.x.s`	Float prefix on extraction
7	`vfmv.s.f vd, zero`	`vmv.s.x vd, zero`	Float move, integer source
8	`vfmax.vf vd, vs, zero`	Insert `fmv.w.x`, fix operand	Integer reg. as float scalar
9	`vfmax.vf vd, zero, vs`	Reorder operands and fix	Reversed operands
10	`vfmax.vf vd, vs, x0`	Replace with `ft0`	`x0` as float operand
11	`addi rd, rs, reg`	`add rd, rs, reg`	Immediate versus register
12	`sub rd, rs, imm`	`addi rd, rs, -imm`	Register-only subtraction
13	`b label`	`j label`	ARM branch mnemonic
14	`mv ft0, zero`	`fmv.w.x ft0, zero`	Integer move to float reg.
15	`sw fN, offset(sp)`	`fsw fN, offset(sp)`	Integer store for float reg.
16	`lw fN, offset(sp)`	`flw fN, offset(sp)`	Integer load for float reg.
17	`vle32.v vd, (r1)(r2)`	Precompute address, then load	Indexed addressing
18	`vsetvli eN`	Add `rd, rs1` operands	Missing required operands

Logging and checkpoint selection. Every candidate is logged with its prompt identifier, checkpoint, raw response, extracted and processed assembly, compilation and execution status, failure category, DEV subtest vector, reward, latency, and token usage. Checkpoints are chosen on the validation tuples, and a checkpoint is kept only if it improves the CR/ER trade-off without losing previously solved operators; test tuples are used only for final reporting.

Records and randomness. All generation and evaluation runs append JSONL records with the fields `model`, `checkpoint`, `operator`, `level`, `shape`, `hardware`, `candidate_idx`, `temperature`, `prompt_hash`, `raw_response`, `asm_hash`, `postprocess_actions`,

`compiled`, `correct`, `failure_type`, `compile_log`, `run_log`, `dev_subtests`, `reward`, `median_time`, and `baseline_time`. Randomness comes from model sampling and provider-side serving, so we store every candidate rather than only aggregate scores, and CR, ER@k, failure rates, and PS are computed from these records without regenerating candidates. Timing uses warmup and repeated runs with the median, which limits the influence of operating-system noise.

E Execution Verification and Reward Details

This appendix expands Section 5.2 with execution processing, rewards, and DEV harnesses.

E.1 Execution Processing and Binary Feedback

Group Relative Policy Optimization (GRPO) (Shao et al., 2024) compares executions. For response y and context x, assembly extractor E and deterministic post-processor P_x produce assembly $\widetilde{a}$:

$$\widetilde{a}(y, x) = P_x(E(y)) \,. \tag{12}$$

On $\widetilde{a}$, binary $c(y, x)$ marks successful extraction/compilation/linking, and $f(y, x)$ full-harness correctness (1: pass; 0: fail); $c = 0$ implies $f = 0$. Correct kernels alone receive hardware timing. Processed code is compiled and executed, and logs retain both raw and processed outputs. Checkpoint selection uses validation data, with final tests held out.

DEV addresses this remaining feedback bottleneck for structured and fused operators such as `sgemm`, `conv2d`, and `attention`. With a binary full-kernel check, two compilable but incorrect programs receive the same correctness reward even when one has useful loops or partial computations. For one context, GRPO samples G candidates indexed by i with rewards r_i, mean $\bar{r}$, and population standard deviation σ_r, and assigns each candidate the advantage $A_i = (r_i - \bar{r})/(\sigma_r + \epsilon)$ with a small constant $\epsilon > 0$. If all rewards equal a common value r_0, both σ_r and every A_i vanish:

$$r_i = r_0 \text{ for all } i \quad \Longrightarrow \quad \sigma_r = 0 \quad \text{and} \quad A_i = 0 \text{ for all } i. \tag{13}$$

Here r_0 is any shared reward value. Thus, the reward-dependent update has no preference among those candidates. DEV makes such groups distinguishable by checking partial semantic progress. This complements SFT: the first stage produces executable candidates, while DEV distinguishes partially successful executions during the second stage.

E.2 Performance Feedback and Reward Semantics

Performance feedback remains conditional on full correctness. Let $m(y, x) \in \{0, 1\}$ indicate that valid candidate and GCC timing measurements are available on the same physical target. On timed, correct candidates, define $s_{\mathrm{GCC}}(y, x) = T_{\mathrm{GCC}}(x)/T_{\widetilde{a}}(x)$, where T_{GCC} and $T_{\widetilde{a}}$ are positive runtimes for the matching GCC baseline and processed candidate under the same task and hardware. The performance term is

$$r_{\mathrm{perf}}(y, x) = \begin{cases} \eta \log_2 s_{\mathrm{GCC}}(y, x), & f(y, x) = 1 \text{ and } m(y, x) = 1, \\ 0, & \text{otherwise}, \end{cases} \qquad \eta = 0.10, \tag{14}$$

where η is the performance scale and $\log_2$ the base-two logarithm, so doubling the speed contributes η, equal speed contributes zero, and larger speedups earn logarithmically larger rewards.

For simple kernels, the reward after successful compilation is $0.10 + 0.70 f(y, x) + r_{\mathrm{perf}}(y, x)$; compilation failure receives -0.20. DEV replaces the binary correctness term by weighted diagnostic progress. The same constants may be denoted $r_{\mathrm{fail}} = -0.20$, $r_{\mathrm{comp}} = 0.10$, $r_{\mathrm{sub}} = 0.70$, and $r_{\max} = 1.0$. The cap applies only when compilation succeeds and every DEV subtest passes. A complete DEV pass without timing earns 0.80, not 1.0. For DEV-trained operators, $r(y, x) = r_{\mathrm{DEV}}(y, x)$; the full harness still determines reported correctness and timing.

The diagnostic family is $\mathcal{H}_o = \{(h_k, w_k)\}_{k=1}^{K_o}$ for operator o, with K_o tests indexed by k. Each outcome $h_k(y, x)$ equals one for a pass and zero for failure or timeout on the processed response to context x. The nonnegative weights w_k sum to one. The all-subtests-pass flag is

$$b_{\mathrm{DEV}}(y, x) = \prod_{k=1}^{K_o} h_k(y, x). \tag{15}$$

The product equals one precisely when every diagnostic passes, including zero-weight diagnostics. Diagnostics run independently. The weighted progress score is $p_{\mathrm{DEV}}(y,x) = \sum_{k=1}^{K_o} w_k h_k(y,x) \in [0,1]$, and neither this score nor the flag b_{DEV} replaces the full-harness predicate f.

The uncapped reward is $\widetilde{r}_{\mathrm{DEV}}(y,x) = -0.20$ when $c(y,x) = 0$ and $\widetilde{r}_{\mathrm{DEV}}(y,x) = 0.10 + 0.70\, p_{\mathrm{DEV}}(y,x) + r_{\mathrm{perf}}(y,x)$ when $c(y,x) = 1$, where $c(y,x)$ is the binary indicator of successful extraction, compilation, and linking. The final reward supplied to GRPO is

$$r_{\mathrm{DEV}}(y,x) = \begin{cases} \min\{1.0, \widetilde{r}_{\mathrm{DEV}}(y,x)\}, & c(y,x) = 1,\ b_{\mathrm{DEV}}(y,x) = 1, \\ \widetilde{r}_{\mathrm{DEV}}(y,x), & \text{otherwise.} \end{cases} \tag{16}$$

The minimum selects the smaller of the uncapped reward and 1.0 only after compilation and every diagnostic pass. Otherwise the uncapped reward is retained; this operation does not turn partial diagnostic success into full-kernel correctness.

E.3 SGEMM Diagnostic Harnesses

For SGEMM diagnostics, $A \in \mathbb{R}^{M\times L}$ and $B \in \mathbb{R}^{L\times N}$ are float32 input matrices, and $C = AB$ is the expected output; M, N, and L are the row, column, and reduction dimensions, respectively, and $\mathbb{R}$ denotes real values represented in floating point. Matrix indices are zero-based: i here denotes a row index, $[0,:]$ a first row, and $[:,0]$ a first column. The symbols I, $\mathbf{1}$, and dot denote an identity matrix, a vector of ones, and a scalar dot product. Identity probes use compatible shapes, with $M = N = L$ for $A = B = I$. In row-major contiguous float32 storage, successive rows of B start $N \times 4$ bytes apart. Each micro-harness constructs diagnostic inputs, calls the kernel, checks selected outputs, and returns a pass/fail exit code from a static binary run under QEMU. H1–H6 are test identifiers whose output probes check the named obligations.

Table 13: Example DEV sub-properties for `sgemm`. The six weights sum to one.

ID	Sub-property	Weight	Test Method
H1	Loop iteration completeness	0.10	All M rows of C written
H2	Column coverage	0.10	All N columns of row 0 written
H3	A-matrix loading	0.15	$B{=}I$: verify $C[i,0]{=}A[i,0]$
H4	Dot product accumulation	0.25	Verify $C[0,0]{=}\mathrm{dot}(A[0,:], B[:,0])$
H5	B-pointer stepping	0.25	$A[0,:]{=}\mathbf{1}$: probe B-row stepping ($N{\times}4$ bytes per reduction step)
H6	Output address computation	0.15	$A{=}I$, $B{=}I$: verify $C{=}I$

E.4 DEV Families Beyond SGEMM

Each DEV micro-harness constructs diagnostic inputs, calls the kernel under test, checks selected output properties, and returns pass or fail through its exit code. The harnesses are compiled with `riscv64-linux-gnu-gcc -static -march=rv64gcv` and executed with `qemu-riscv64`; one reward query costs about 200 ms of compilation and 50 ms for six QEMU runs. Table 14 lists the sub-properties used for the other structured and fused operator families.

E.5 Why Partial Progress Provides a Learning Signal

Let $r_{\mathrm{sub}} = 0.70$ be the subtest progress scale. For two compilable responses y and y' to the same prompt, with equal performance terms, their uncapped reward difference is

$$\widetilde{r}_{\mathrm{DEV}}(y,x) - \widetilde{r}_{\mathrm{DEV}}(y',x) = r_{\mathrm{sub}} \sum_{k=1}^{K_o} w_k \big[h_k(y,x) - h_k(y',x)\big], \tag{17}$$

where y' denotes the comparison response. If both fail the full harness, their performance terms are zero; improving a positive-weight subtest without losing another increases this reward even though binary correctness is unchanged. For example, a candidate passing only H1 and H2 has

Table 14: DEV sub-properties for further operator families.

Operator family	Sub-property	Diagnostic check
GEMM / SGEMM	Accumulator initialization, inner-loop coverage, and tail-safe stores	Identity and one-hot matrices isolate whether rows, columns, and reduction axes are visited exactly once
Softmax	Max subtraction, exponent and sum consistency, and row-wise normalization	Rows with large offsets and repeated maxima check numerical stability and reduction finalization
LayerNorm	Mean and variance separation and epsilon handling	Constant, nearly constant, and high-variance rows separate variance bugs from affine bugs
Attention / MHA	Score computation, masking, softmax, and value aggregation across heads	Synthetic masks and single-active-token inputs expose stage-order and pointer-stride mistakes
Fused kernels	Dependency preservation across producer and consumer stages	Inputs at activation or normalization boundaries verify that intermediate values are consumed before being overwritten

$p_{\mathrm{DEV}} = 0.20$ and reward 0.24 without performance, compared with 0.10 for a compilable candidate passing none. Such differences create within-group preferences even when binary correctness is unchanged. The vector of subtest outcomes also makes failures interpretable and supplies diagnostic hints for later generation or repair, so each DEV query yields both a dense reward and a precise description of what a kernel already does correctly on the target Rvv hardware under test.

F Additional Experimental Results

This appendix follows the organization of Section 6 and extends each of its research questions.

F.1 Ablation Study and Comparative Analysis (RQ2)

As shown in Figure 11, we extend the ablation of Figure 5 with a fourth variant that removes the ISA safeguards used during generation.

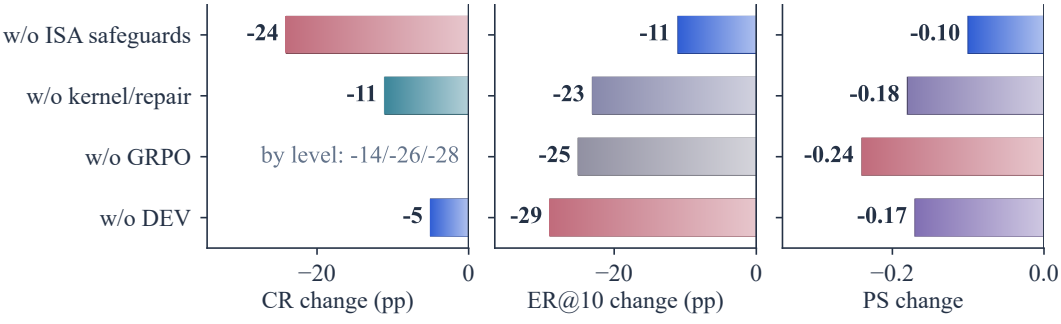
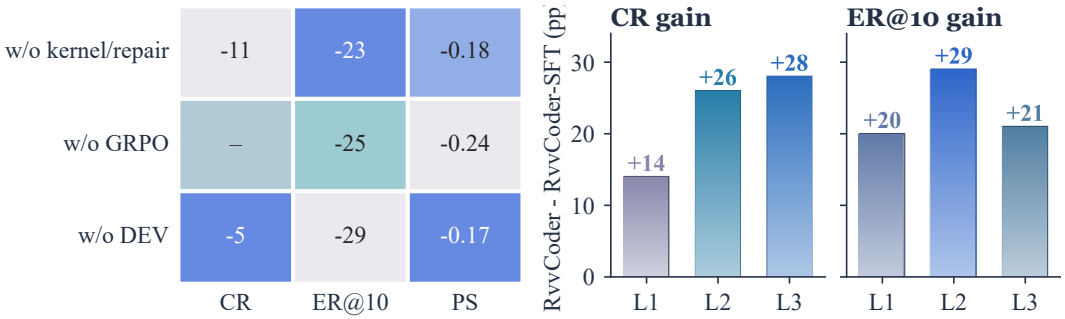


Figure 11: Changes relative to full RvvCoder for four ablated variants.

Every safeguard contributes. Removing the ISA safeguards lowers CR by 24 points, ER@10 by 11 points, and PS by 0.10, so these safeguards mainly protect compilation. Removing the GRPO stage lowers CR by 14, 26, and 28 points on Levels 1–3, with the largest losses on the harder levels.

The second stage helps most on harder levels. As shown in Figure 12(b), the second stage adds 14, 26, and 28 points of CR and 20, 29, and 21 points of ER@10 on Levels 1–3. Figure 12(a) shows that each variant loses on every metric, with the largest ER@10 loss without DEV.

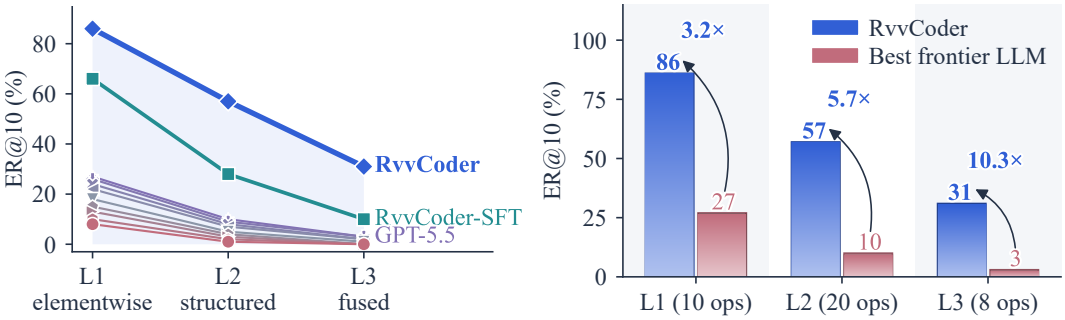


(a) Changes on CR, ER@10, and PS (b) RvvCoder minus RvvCoder-SFT

Figure 12: Ablation details. (a) Metric grid of three variants. (b) Second-stage gains by level.

F.2 Analysis of ISA-Constrained Task Difficulty (RQ3)

As shown in Figure 13, we analyze difficulty and the advantage of RvvCoder across levels.



(a) ER@10 of all models by level (b) RvvCoder versus the best frontier LLM

Figure 13: Task difficulty across levels. (a) ER@10 per model. (b) Advantage of RvvCoder.

Difficulty and advantage across levels. From Level 1 to Level 3, the best frontier ER@10 falls from 27% to 10% and 3%, while RvvCoder retains 86%, 57%, and 31%. Its advantage over the best frontier LLM therefore grows from 3.2× to 5.7× and 10.3×. Structured reuse and fused dependencies are the hardest parts of the task, and they are where RvvCoder helps most.

Table 15: Candidate-level diagnostic outcomes on RvvBench (%).

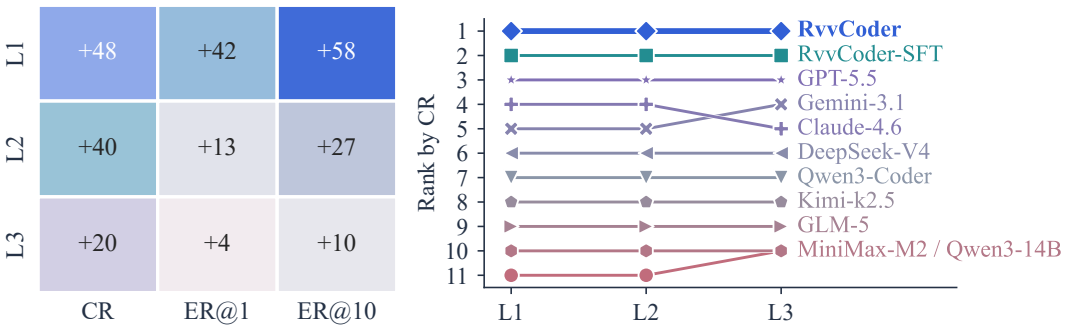
Model	No ASM	ISA/ABI	Compile failure	Wrong	Pass
Qwen3-14B	18	34	29	17	2
RvvCoder-SFT	6	16	21	41	16
RvvCoder	2	5	8	43	42

Failures concentrate on the task constraints. As shown in Table 15, 81% of the candidates of the untuned Qwen3-14B fail before execution: 18% produce no usable assembly, 34% violate ISA/ABI rules, and 29% fail compilation, while only 2% pass. Both constraint families of the task, ISA/ABI legality and functional semantics, are therefore active bottlenecks.

F.3 Analysis of RvvBench Supervision and Evaluation (RQ4)

As shown in Figures 14 and 15, we analyze RvvCoT gains by level and suite discrimination.

RvvCoT gains by level. As shown in Figure 14(a), SFT on RvvCoT raises CR by 48, 40, and 20 points, ER@1 by 42, 13, and 4 points, and ER@10 by 58, 27, and 10 points on Levels 1–3. The



(a) RvvCoT SFT gain (pp)

(b) Model ranks by CR

Figure 14: Benchmark analysis. (a) RvvCoder-SFT minus Qwen3-14B. (b) CR ranks by level.

gains are largest on Level 1, where ER@10 rises from 8% to 66%, and still reach the fused Level-3 operators, whose CR grows from 4% to 24%.

The evaluation suite discriminates without saturating. As shown in Figure 15, ER@10 spans 78, 56, and 31 points on Levels 1–3 across the 11 models, and CR spans 62, 66, and 48 points. RvvBench thus separates models clearly at every level. Rankings are also stable, as Figure 14(b) shows: Kendall's τ between the level-wise CR rankings is at least 0.95.

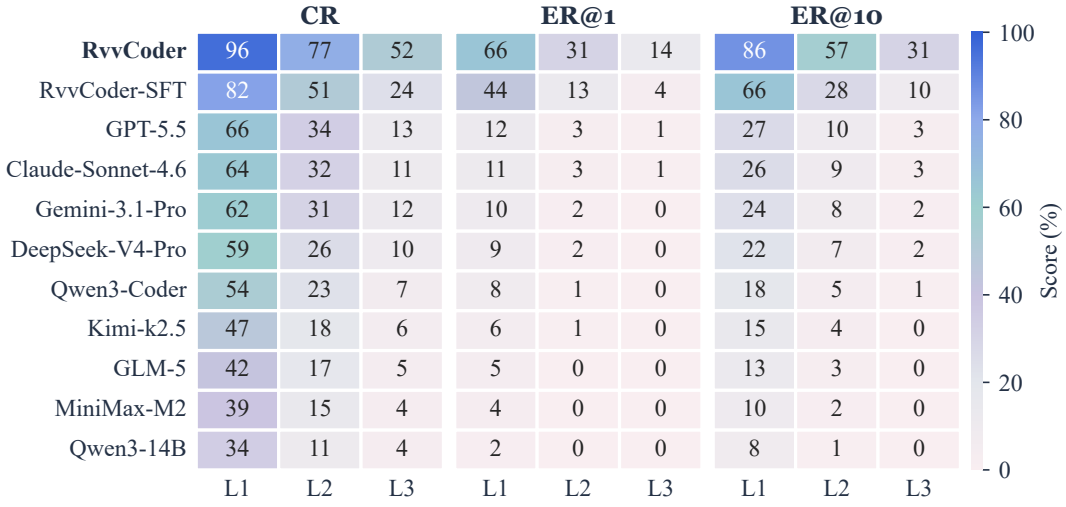


Figure 15: Scores of all models from Table 1 on a shared 0–100 color scale.

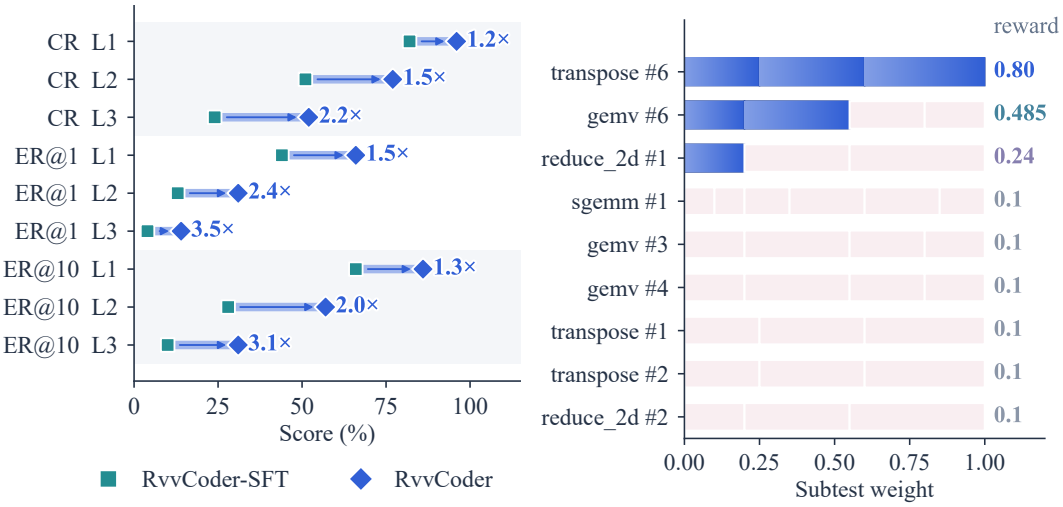
F.4 Analysis of Execution-Grounded RL with DEV (RQ5)

As shown in Figures 16 and 17, we analyze second-stage gains, DEV learning speed, and rewards.

Second-stage gains by level. As shown in Figure 16(a), GRPO with DEV improves every metric at every level. ER@10 increases by 1.3×, 2.0×, and 3.1× on Levels 1–3, and ER@1 by 1.5×, 2.4×, and 3.5×, so the largest gains occur on the structured and fused operators DEV targets.

DEV credits partial progress. Figure 16(b) lists the subtests passed by the compiled development candidates of Figure 8(b). A transpose candidate passes all subtests and earns 0.80, a GEMV candidate passing two of four earns 0.485, and a 2-D reduction candidate passing one of three earns 0.24, compared with 0.10 for compiled candidates that pass none.

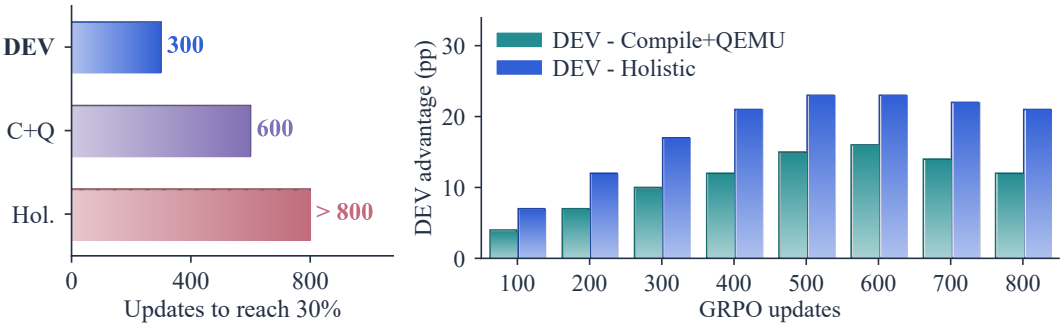
DEV leads at every checkpoint. As shown in Figure 17, DEV needs 300 updates to reach 30% ER@10, against 600 for Compile+QEMU and more than 800 for the holistic reward. It leads Com-



(a) RvvCoder-SFT to RvvCoder

(b) DEV subtests of compiled candidates

Figure 16: The second stage. (a) Ratios per metric and level. (b) Weighted subtests passed.



(a) Updates to reach 30%

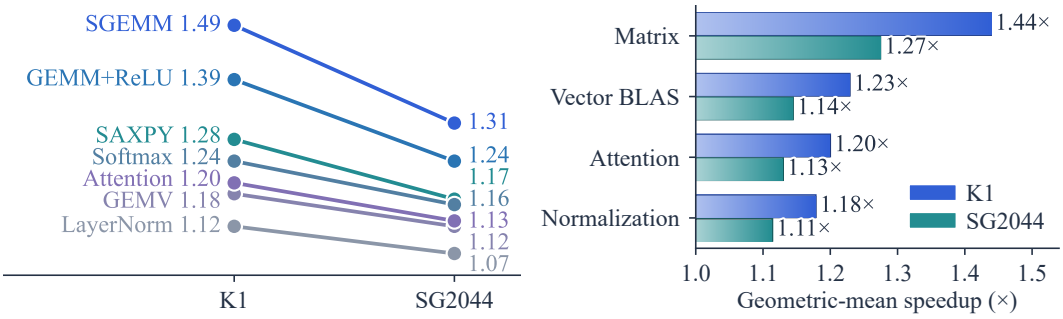
(b) DEV lead at each checkpoint

Figure 17: Reward design. (a) Updates to the 30% target. (b) Lead of DEV in ER@10.

pile+QEMU by 4 to 16 points and the holistic reward by 7 to 23 points at every checkpoint, and averaged over the 800 updates its ER@10 is 34.8%, versus 24.3% and 17.8%.

F.5 Real-Hardware Efficiency (RQ6)

As shown in Figures 18 and 19, we analyze speedup consistency across boards and classes.

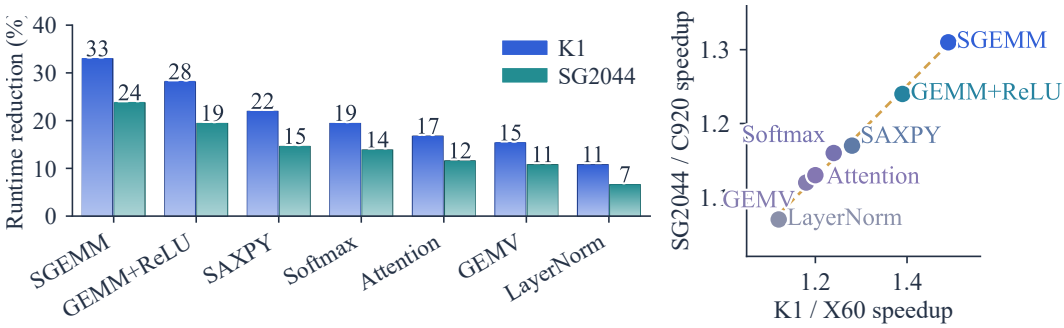


(a) Operator speedups on both boards

(b) Speedup by operator class

Figure 18: Hardware analysis. (a) Cross-board ordering. (b) Class-wise geometric means.

Gains transfer across boards and classes. As shown in Figure 18(a), the seven operators keep the same speedup ordering on both boards, with Spearman $\rho = 1.00$. Matrix kernels benefit most, with geometric means of $1.44\times$ and $1.27\times$, followed by vector BLAS at $1.23\times$ and $1.14\times$, attention at $1.20\times$ and $1.13\times$, and normalization at $1.18\times$ and $1.11\times$.



(a) Runtime reduction per operator

(b) Speedups on the two boards

Figure 19: Hardware analysis. (a) Runtime saved against the baseline. (b) Cross-board scatter.

Every operator saves runtime. As shown in Figure 19(a), RvvCoder saves 11% to 33% of runtime on K1/X60 and 7% to 24% on SG2044/C920, led by SGEMM and GEMM+ReLU. Figure 19(b) shows that the per-operator speedups of the two boards are almost collinear, with Pearson $r = 0.997$.

F.6 Case Study (RQ7)

As shown in Figure 20, the SGEMM case follows the full pipeline from constraints to speedups.

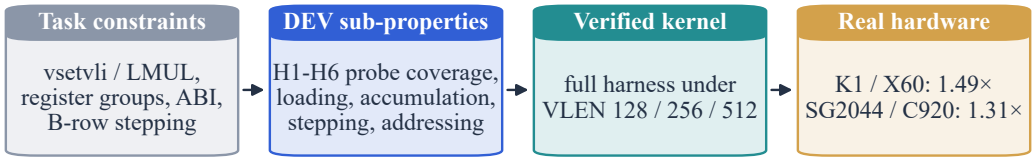


Figure 20: The SGEMM case from task constraints to the measured speedups on both boards.

Case 1: cross-ISA interference. A candidate opens with the ARM directive `.syntax unified`, which the RISC-V assembler rejects at line 1. Syntax from better-represented ISAs leaks into Rvv code when domain data are scarce. The ISA-rule layer of RvvCoT supplies Rvv-only directives, and ISA/ABI violations fall from 34% for the base model to 16% after SFT and 5% for RvvCoder.

Case 2: illegal vector configuration. `vsetvli t1, s3, e32, ta, ma, m8` places `LMUL` after the tail and mask policies, so the assembler reports illegal operands; the `vtype` fields must follow the order `SEW`, `LMUL`, tail policy, and mask policy. ISA rules and verified repairs target such errors, and pre-execution failures fall from 81% to 43% and 15%.

Case 3: legal code, wrong result. An FMA kernel loads a, b, and c into `v8`, `v16`, and `v24` and then issues `vfmacc.vv v8, v16, v24`, which accumulates into a and computes $a + b \cdot c$. The program assembles, links, and runs, so only execution against the reference exposes the error, and execution-grounded GRPO raises passing candidates from 2% to 16% and 42%.

Case 4: partial SGEMM progress. A row kernel uses `vl`-driven strip-mining that writes every output element and keeps a vector accumulator across the reduction, but it advances B by $4\,$`vl` bytes instead of the row pitch of $4N$ bytes for N columns. A binary check scores it like any failure, whereas DEV probes coverage and accumulation separately from B-pointer stepping (H5).

The final SGEMM kernel. As shown in Table 16, the final kernel places `vsetvli` outside inner loops when the configuration is unchanged, keeps vector accumulators live across the reduction, and computes byte-address increments in scalar registers before vector memory operations. These

choices reduce repeated setup and improve register reuse, and the verified kernel runs $1.49\times$ and $1.31\times$ faster than the strongest applicable baseline on the two boards.

Table 16: Implementation comparison for the SGEMM case on SpacemiT K1.

System	Implementation approach	Design consideration
GCC -O3	Compiler vectorization with compiler-selected `vsetvli` placement and accumulator allocation	Vector setup and register reuse depend on the generated loop structure
TVM Rvv	Template-driven tiling and schedule selection	Available schedules determine the explored implementation choices
OpenBLAS Rvv	Expert-tuned micro-kernel for covered BLAS shapes	Provides specialized implementations where the BLAS interface matches
RvvCoder	`LMUL`-aware strip-mining, live vector accumulators, and explicit B-pointer stepping	Couples vector configuration and data movement with the correctness requirements of the operator

G Limitations and Future Work

Evaluation scope. RvvBench covers 38 float32 and int32 operators; quantized int8 and int16 kernels, sparse operators, and graph kernels are not yet included. Correctness is established by a finite harness under three `VLEN` settings rather than by formal verification, so a kernel that passes every test may still fail on untested inputs. Tuple-level separation keeps exact test tuples out of training, but primitive operator families may appear in both, so it is not a test of entirely unseen operators.

Hardware scope. Real-hardware speedups are measured for seven operators on two boards, and the performance summary covers only tasks with correct, stably timed candidates. Library baselines enter only where they match the task semantics, so the reference baseline varies by operator. The training reward measures speed against GCC, while evaluation uses the strongest baseline.

Evidence scope of the analyses. The reward-resolution analysis uses 32 development candidates from four Level-2 operators. The reward-design curves start from one shared checkpoint that differs from the reported RvvCoder-SFT, and the candidate-level outcome table pools all candidates included for each model in the failure analysis. Case 4 is annotated by reading the program.

Method dependencies. DEV requires executable sub-properties designed for each operator family, and passing all of them does not imply whole-kernel correctness. RvvCoT relies on teacher LLMs for explanations and repairs, filtered by execution. The post-processor fixes surface errors that the model should ideally avoid on its own, and QEMU and hardware verification make every reward query more expensive than a text-only reward.

Future work. We plan to extend RvvBench to quantized, sparse, and graph kernels, to other vector ISAs such as Arm SVE and LoongArch LSX, and to more RISC-V boards. Automatically deriving DEV sub-properties from operator specifications, bringing hardware timing into the training loop more directly, and combining generation with formal equivalence checking are further directions toward reliable and efficient kernel synthesis.