A SELF-EVOLVING EVALUATION METHODOLOGY FOR NPU OPERATORS WITH THEORY-DERIVED PERFORMANCE UPPER BOUNDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Deep neural network inference now spans edge devices and data centers, and its matrix- and vector-intensive computation calls for dedicated NPUs. Running a network on such an accelerator ultimately comes down to its low-level operators, whose quality directly determines model performance. Although operators can now be written in many ways, there remains no unified standard for whether an operator is correct and efficient. We therefore propose an NPU-oriented operator benchmark whose tasks are designed around the NPU's matrix/vector units and multi-level memory hierarchy, spanning a difficulty spectrum from vector-only operators to fused matrix-vector pipelines. However, fixed cases are insufficient for truly evaluating operators, as they capture only local behavior and can drive optimization into a local optimum. To address this, we propose Self-Evolving Test-Case Generation (SETG), which dynamically generates cases around two kinds of risk, namely operator numerical errors and NPU execution paths, while continuously refining them near anomalies based on real-device feedback. For evaluation, to determine how far each operator still is from its hardware limit, we define Hardware-Attainable Performance (HAP), which analytically derives the performance ceiling from NPU scheduling characteristics, measures both achieved optimization and remaining headroom, and identifies the limiting unit to guide optimization. Our experiments show that the benchmark both discovers real problems and aids effective optimization. On the one hand, in bare base-model evaluation, only four of eight base models obtain nonzero scores on all eight operators, and the best average score is 79.8, with many submissions failing on compilation or correctness, showing that the benchmark exposes real operator problems. On the other hand, in a controlled iterative-optimization study, about 70% of operators score higher on hidden cases than on public cases, with average score up by 2.5 points and average speedup up by 0.51×, indicating that the optimization yields a genuine quality gain that generalizes to unseen cases.

1 INTRODUCTION

Deep neural network inference now spans edge devices to data-center services, and processing its matrix- and vector-intensive computation requires dedicated accelerators rather than general-purpose GPUs (Liao et al., 2021). On an NPU, every network ultimately executes as a set of low-level operators, whose quality directly determines model performance (Ouyang et al., 2025). Writing such operators, however, is inherently difficult, because developers must coordinate heterogeneous matrix and vector units, explicitly manage a multi-level memory hierarchy (Liao et al., 2021), and satisfy strict alignment constraints (Zhang et al., 2026).

Large language models and code-generation agents are increasingly used to write these operators (Ouyang et al., 2025; Li et al., 2025). Yet operators are hard to evaluate, because a fixed set of cases reflects only local behavior and can drive optimization into a local optimum (Lange et al., 2025), so there is still no unified standard for whether a generated operator is correct and efficient on an NPU.

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.

Existing kernel benchmarks target CUDA, Triton, or server GPUs (Lin et al., 2026), and the few that include NPU targets either treat the NPU as one backend among many (Wen et al., 2025) or bundle the task set with a specific generation framework (Cao et al., 2026). A mechanical port of GPU benchmarks to an NPU therefore fails, because their execution models and memory hierarchies differ fundamentally. We therefore introduce an NPU-oriented operator benchmark whose tasks are designed around the NPU's matrix/vector units and multi-level memory hierarchy, spanning a difficulty spectrum from vector-only operators to fused matrix-vector pipelines.

On top of this task set, this paper proposes two evaluation mechanisms as its core contribution. The first, Self-Evolving Test-Case Generation (SETG), keeps the suite growing beyond fixed cases. The second, Hardware-Attainable Performance (HAP), scores operators against a theory-derived physical limit.

To keep fixed cases from driving optimization into a local optimum, SETG generates cases around operator numerical errors and NPU execution paths and refines them from real-device feedback. To determine how far each operator still is from its hardware limit, HAP derives the limit from NPU scheduling characteristics (Williams et al., 2009), measures both achieved optimization and remaining headroom, and identifies the unit that limits each operator. Together, these two mechanisms turn the task set into an evaluation system that discovers, guides, and measures.

In summary, this paper presents an NPU-oriented operator benchmark whose core contributions are two evaluation mechanisms: Self-Evolving Test-Case Generation (SETG), which generates and refines test cases from real-device feedback, and Hardware-Attainable Performance (HAP), which scores kernels against a theory-derived physical ceiling. Our evaluation shows that the benchmark discovers real operator problems and guides effective optimization.

2 Related Work

Kernel generation methods. Kernels were traditionally written through hand-tuned libraries and auto-tuning compilers, which demand substantial human effort and are tied to specific hardware. Large language models, reinforcement learning, and agentic search have recently automated kernel generation. CUDA Agent applies large-scale agentic RL to high-performance CUDA kernel generation (Dai et al., 2026). Kevin trains a multi-turn RL model to generate CUDA kernels (Baronio et al., 2025). EvoEngineer decomposes LLM-based CUDA kernel code evolution into traversal and population management (Guo et al., 2025). AutoKernel optimizes GPU kernels through an autonomous iterative agent loop (Jaber & Jaber, 2026). GEAK combines a Triton kernel agent with dedicated evaluation benchmarks (Wang et al., 2025). These methods advance rapidly, yet each is evaluated on its own task set and tied to a specific generation recipe, so their results are not comparable and no shared yardstick exists for NPU kernels.

GPU kernel benchmarks. KernelBench established compilation and speedup protocols for CUDA kernels (Ouyang et al., 2025). TritonBench evaluates LLM-generated Triton operators against real-world GPU workloads (Li et al., 2025). SOL-ExecBench scores kernels against speed-of-light hardware limits (Lin et al., 2026). ENAMEL measures the efficiency of LLM-generated code (Qiu et al., 2025). ConvCodeWorld benchmarks conversational code generation in reproducible feedback environments (Han et al., 2025). These benchmarks, however, are anchored to CUDA, Triton, or server GPUs, rely on fixed task sets, and score against software baselines.

NPU and other accelerator benchmarks. NPUEval evaluates LLM-generated NPU kernels on AMD hardware (Kalade & Schelle, 2025). KernelCraft benchmarks agentic close-to-metal kernel generation on emerging accelerators (Nie et al., 2026). MultiKernelBench spans multiple platforms at the cost of per-platform depth (Wen et al., 2025). MobileKernelBench evaluates mobile CPU kernels without cube and vector concurrency (Zou et al., 2026). AscendKernelGen couples its task set to a generation framework (Cao et al., 2026). AscendCraft transcompiles kernels through a DSL-guided pipeline (Wen et al., 2026). AscendOptimizer optimizes operators through an episodic agent (Wu et al., 2026). EvoKernel synthesizes NPU kernels through value-driven memory (Zheng et al., 2026). None of these provides a versioned benchmark that reflects the Ascend vector and matrix programming model and a shared hardware-grounded evaluation scale.

In contrast, our benchmark contributes two mechanisms that close these gaps. The first, SETG, keeps the suite from becoming a static dataset by generating cases around operator numerical errors

Table 1: Related benchmarks and the proposed NPU benchmark.

Benchmark	Target	Performance score	Anti-reward-hacking	Maintenance
KernelBench (Ouyang et al., 2025)	NVIDIA GPU	Speedup	No	Community
TritonBench (Li et al., 2025)	NVIDIA GPU	Speedup	No	Community
SOL-ExecBench (Lin et al., 2026)	NVIDIA GPU	Speed-of-Light	Partial	Community
MultiKernelBench (Wen et al., 2025)	GPU/NPU/TPU	Speedup	No	Academic
MobileKernelBench (Zou et al., 2026)	Mobile CPU	Speedup threshold	No	Academic
Ours	**Ascend NPU**	**HAP hardware limit**	**Yes**	**Versioned platform**

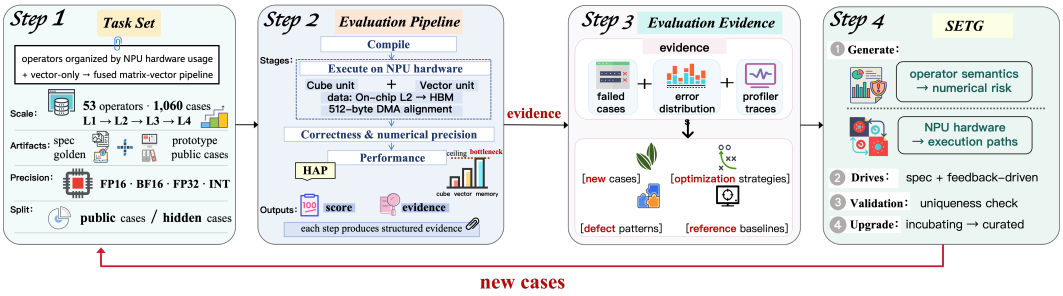

Figure 1: Overview of the benchmark. The task set is evaluated through a pipeline scored by HAP, and evaluation evidence flows back through SETG to generate new cases.

and NPU execution paths and refining them from real-device feedback. The second, HAP, scores kernels against a theory-derived physical hardware limit, measuring how close each kernel remains to the hardware ceiling. Table 1 summarizes these differences, and to our knowledge this is the first NPU-oriented operator benchmark to combine self-evolving test cases with a theory-derived hardware limit.

3 Benchmark Design

3.1 Overview

We propose an NPU-oriented operator benchmark organized around the NPU's matrix and vector units and its multi-level memory hierarchy. Figure 1 shows the overall architecture, which comprises a task set, an evaluation pipeline, and a feedback loop. A submitted operator is compiled against the target NPU's programming model, verified for functional and numerical correctness, and scored for performance against a hardware limit derived from the NPU's computing and memory throughput. The evaluation evidence, including failed cases, error distributions, and profiler traces, then flows back to generate new test cases. Two mechanisms drive this process. SETG generates and evolves the test cases, and HAP scores performance against the NPU's hardware limit. Sections 4 and 5 detail these mechanisms, and the rest of this section presents the task set and the evaluation pipeline.

3.2 Task set

The task set defines what each operator is evaluated against. It contains 53 operators and 1,060 public test cases spanning vector-only kernels to fused matrix-vector pipelines, distributed as 8/16/21/8 across the four levels, which covers the spectrum of how operators use the NPU's computing units. Each operator ships with a structured specification, a machine-readable prototype, public test cases, and a golden reference implementation. The specification and prototype drive SETG's case generation, and the golden reference anchors the correctness check. Each operator keeps 20 public cases for local iteration and 80 hidden cases reserved for leaderboard evaluation, which prevents submitters from overfitting to disclosed inputs. The task set also covers emerging custom operators such as DeepSeek Engram (Cheng et al., 2026), with precision spanning FP16, BF16, FP32, and integer paths, and lower-precision formats left as extensions (Darvish Rouhani et al., 2023; Tseng et al., 2025). Beyond the initial suite, the repository maintains an incubating area with 59 additional

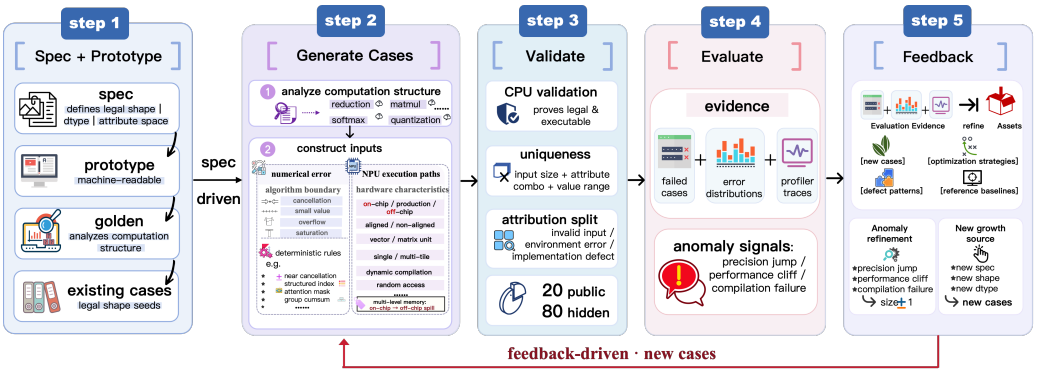

Figure 2: Self-evolving test-case generation. SETG generates cases from operator specifications along two coverage axes, numerical error and NPU execution paths, and evaluation evidence flows back to keep the suite growing.

operators and 1,202 cases, so the generation pipeline continues to produce new operator and case families.

3.3 Evaluation Pipeline

The evaluation pipeline realizes this flow in a unified environment. Each submission is packaged as a standard operator project, and the platform records the four evidence dimensions and generates a report for each submission. Performance is measured at the kernel level with a hardware profiler after warmup, cache eviction, and process isolation, which excludes high-level library and system-call overhead from the reported time. Results are indexed by model, agent, configuration, and timestamp, and performance baselines and hardware limits are stored in versioned hardware metadata, so the same task can be compared across versions and baselines can be refreshed when the software stack changes without rewriting the task specification. The underlying logs and reports remain available for reproducibility.

4 Self-Evolving Test-Case Generation

A fixed set of cases can only reflect an operator's behavior on a specific set of inputs, so it cannot cover the operator's real numerical errors and execution paths, and it can be gamed by optimization algorithms. We therefore introduce Self-Evolving Test-Case Generation (SETG), which generates cases automatically from each operator's specification and machine-readable prototype and keeps refining them from real-device feedback, so the suite keeps growing.

As shown in Figure 2, SETG generates cases from operator specifications along two coverage axes, numerical error and NPU execution paths, and evaluation evidence flows back to produce new cases. This section describes specification-driven case generation, case validation and the public/hidden split, and the self-growth feedback loop.

4.1 Specification-Driven Case Generation

Cases are generated automatically from each operator's specification and machine-readable prototype. The generator first identifies the operator's computation structure from the golden reference, then constructs inputs along two axes, numerical error and NPU execution paths.

For numerical error, the generator maps each computation structure to the inputs where error concentrates. Matrix multiplication and reduction accumulate error over a long reduction axis, so the generator constructs long-K inputs. Normalization divides by a near-zero quantity, so it constructs small-value inputs. Softmax can overflow its exponential and underflow its probabilities, so it con-

structs large-logit inputs. Quantization saturates its codes and loses scale precision, so it constructs extreme code inputs. Some risks require two inputs to satisfy a specific relation. For cancellation, the generator sets $b \approx -a$ deterministically so that two large opposite-signed values actually cancel.

For NPU execution paths, the generator covers three input scales that fit on-chip, production, and off-chip memory, and both aligned and non-aligned access paths (Liao et al., 2021), so each case exercises a different part of the NPU's memory hierarchy.

4.2 Case validation and public/hidden split

Every case passes two checks before entering the suite. The first is a CPU check: the case runs on the CPU reference implementation to prove that it is legal and executable, so problems later exposed on the NPU can be attributed to the operator implementation. The second is a uniqueness check: each case has a unique combination of input size, attributes, and value range, which avoids duplicate cases diluting coverage. After passing the checks, each operator keeps 20 public cases for local iteration and 80 hidden cases reserved for leaderboard evaluation, which prevents submitters from overfitting to disclosed inputs.

4.3 Self-growth feedback loop

The suite keeps growing beyond its initial cases through two sources. The first source is evaluation feedback. Failed cases, error distributions, and profiler traces exposed during evaluation flow back to the generator and become new cases and refined input conditions. When these failures manifest as a precision jump, a performance cliff, or a compilation failure, the generator produces finer cases near the observed anomaly to approach the error boundary. The second source is new operator specifications and new shape and dtype combinations, which keep generating new cases. New operators first enter an incubating area and are promoted into the curated suite after passing all validation rules. In this way, the suite keeps growing and stays covered against real defects.

5 Hardware-Attainable Performance

Performance is a core dimension for measuring operator implementation quality. How far a kernel remains from its physical performance ceiling directly reflects its optimization quality. We introduce Hardware-Attainable Performance (HAP), which derives each operator's physical performance ceiling and measures both the optimization achieved and the headroom that remains. The per-unit derivation of this ceiling also identifies the unit that limits each operator, which guides where to optimize. This section describes the three-axis score, then the HAP score and its derivation, and finally the correctness and measurement safeguards.

5.1 Three-axis scoring

Each submission's total score is formed by three weighted axes: compilation, correctness, and performance, where the correctness axis covers both functional and numerical correctness. Let δ_{pass} mark compilation, $\delta_{\text{acc},i}$ mark whether case i passes the correctness check, and score_i be its performance score. The operator score is

$$\text{Score}_{\text{op}} = \delta_{\text{pass}} \cdot \left[w_c + \frac{1}{n} \sum_{i=1}^{n} \delta_{\text{acc},i} \big(w_f + w_p \, \text{score}_i \big) \right] \cdot 100. \tag{1}$$

The three axes use weights $w_c = 0.2$, $w_f = 0.3$, and $w_p = 0.5$. This design ensures that a kernel that fails to compile receives no credit, and a case that fails correctness loses only its own credit without affecting other cases.

5.2 Hardware-Attainable Performance

HAP scores the performance axis and measures how far a kernel remains from its physical performance ceiling. Because this ceiling is determined by the operator's own compute and byte counts

and by hardware parameters, it is stable and comparable across operators and lets the HAP score fairly separate the optimization quality of different submissions.

HAP places a kernel's runtime between two reference values: a baseline time $T_{\text{baseline},i}$ and a hardware-limit time $T_{\text{HW},i}$. The score is

$$\text{score}_i = \frac{T_{\text{baseline},i} - T_{\text{HW},i}}{\big(T_{\text{cand},i} - T_{\text{HW},i}\big) + \big(T_{\text{baseline},i} - T_{\text{HW},i}\big)}, \tag{2}$$

where $T_{\text{cand},i}$ is the candidate kernel's runtime. The score equals 0.5 at the baseline, approaches 1 as the kernel approaches the hardware limit, and decays toward 0 as the kernel gets slower. The score is bounded and therefore comparable across operators.

The hardware limit is derived by a Unit-Peak Overlap Bound (Williams et al., 2009). We first decompose the NPU pipeline into execution units, including cube, vector, and memory-transfer engines, and assume that every unit runs at its peak rate and that all units overlap perfectly. The slowest unit is then the bottleneck, and the hardware limit is the maximum of four terms:

$$T_{\text{HW}} = \max\big(t_{\text{read}},\, t_{\text{write}},\, t_{\text{cube}},\, t_{\text{vector}}\big), \tag{3}$$

where the four terms are obtained by dividing the corresponding workload by the bandwidth or throughput:

$$\begin{gathered} t_{\text{read}} = \frac{B_{\text{read}}}{\text{HBM_BW}}, \qquad t_{\text{cube}} = \frac{F}{\text{cube_peak}(\text{dtype})}, \\ t_{\text{vector}} = \frac{N_{\text{vec}}}{\text{vector_rate}(\text{kind}, \text{dtype})} \end{gathered} \tag{4}$$

where B_{read} and B_{write} are the input and output byte counts, F is the floating-point count, and N_{vec} is the number of vector-side element operations. The write term additionally accounts for the on-chip L2 capacity:

$$t_{\text{write}} = \begin{cases} \frac{B_{\text{write}}}{\text{L2_BW}}, & B_{\text{write}} \le C_{\text{L2}}, \\ \frac{C_{\text{L2}}}{\text{L2_BW}} + \frac{B_{\text{write}} - C_{\text{L2}}}{\text{HBM_BW}}, & \text{otherwise}, \end{cases} \tag{5}$$

where C_{L2} is the on-chip L2 capacity, and writes that exceed it spill to high-bandwidth memory. For matrix multiplication, $F = 2MNK$, the read volume is about $MK + KN$, and the write volume is about MN. Each case records its bottleneck class among memory transfer, vector computation, and cube computation, so the score also reveals which unit limits the operator and where optimization would help most. This limit is the fastest time a kernel can reach under ideal overlap, and the HAP score then measures the optimization achieved against it and the headroom that remains.

5.3 Correctness and measurement safeguards

Functional correctness is checked against a high-precision golden reference. In normal value regions, the elementwise relative error is defined as

$$e_i = \frac{|\hat{y}_i - y_i|}{|y_i| + 10^{-7}}, \tag{6}$$

and a tensor passes when both the mean and the maximum of e_i fall below dtype-specific thresholds. In small-value and cancellation regions, relative error is inflated, so we switch to an absolute-error check, counting errors as

$$\text{ErrorCount} = \sum \mathbf{1}\big(|g_t| < \tau_s \ \wedge\ |n - g_t| > \epsilon_s\big), \tag{7}$$

Table 2: Model and agent results on the eight entry-level operators.

Model / Agent	Provider	Nonzero operators	Macro average	20/20 trajectories	Std. dev.
DeepSeek V4.1 Flash	DeepSeek API	8	79.8	23	0.93
Kimi K3	OpenCode Go	8	75.6	23	5.65
GLM-5.3	Zhipu BigModel	8	62.9	17	11.69
Qwen3.8-Max	QwenCloud	8	43.4	7	10.72
DeepSeek V4 Flash	OpenCode Go	6	34.8	8	15.00
DeepSeek V4 Pro	DeepSeek API	7	19.6	0	9.07
MiniMax-M3	MiniMax	4	18.4	3	11.78
MiMo-V2.5-Pro	Xiaomi MiMo	5	9.6	0	10.98

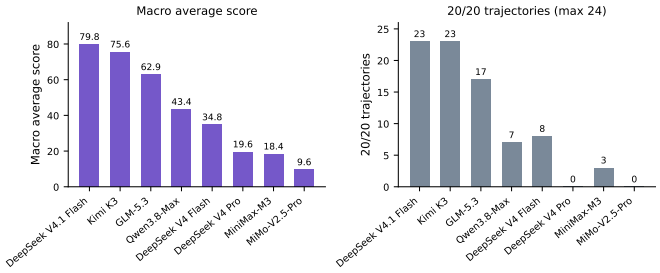

Figure 3: Results for eight models and agents. Left: macro-average score. Right: number of 20/20 trajectories out of 24.

and comparing the candidate's error count with that of a same-precision CPU reference; the candidate passes when its count does not exceed twice the reference count.

Additionally, the measurement process includes warmup, cache eviction, process isolation, and kernel-level profiler timing (Huawei Ascend, 2023), which excludes high-level library and system-call overhead from the reported time. Submission rules prohibit high-level library delegation, CPU fallback, output caching, and timing-API tampering (Lange et al., 2025).

6 Experiments

6.1 Benchmark discovers real problems

To show that the benchmark exposes real operator problems, we evaluate eight base models and agents on the benchmark's entry-level operators, which already expose substantial compilation, correctness, and numerical failures across base models. Table 2 reports the results. Only four of the eight models obtain nonzero scores on all eight operators, the best macro-average score is 79.8, and the weakest model scores only 9.6. This eight-fold score span shows that the benchmark separates strong models from weak ones. Compilation failure is the most frequent failure category, followed by correctness and format failures, which indicates that base models struggle with the Ascend programming model itself. Table 3 reports operator-level difficulty: Exp obtains an average score of 62.7 while ForeachNorm obtains only 28.6, a two-fold gap that shows the benchmark spans both easy and hard operators. This base-model evaluation focuses on the entry-level tier, while the self-evolving and HAP experiments below cover the full suite.

Figure 3 compares the macro-average score and the number of fully successful trajectories. The results separate models that obtain nonzero scores on all operators from models that fail on a substantial subset, and the trajectory standard deviation identifies models whose success is unstable across runs.

Table 3 reports operator-level difficulty and failure categories. Operators such as Exp, Mish, and SwiGLU achieve higher average scores and more fully successful trajectories, whereas ForeachNorm and MaskedScale are consistently harder. This operator-level view shows that the benchmark

separates easy operators from hard ones and reveals model-specific failure patterns that aggregate scores cannot capture.

Table 3: Operator-level difficulty and failure categories.

Operator	Avg. score	Nonzero models	20/20 traj.	Compile	Correct	Format	Device
Exp	62.7	7	13	6	1	3	0
Mish	50.1	6	13	7	3	2	1
SwiGLU	49.9	8	12	6	5	2	1
ForeachAddCdivScalar	44.3	7	11	8	6	2	0
Sigmoid	43.4	7	10	8	3	1	0
Gelu	36.9	8	6	6	5	2	1
ForeachNorm	28.6	4	8	7	2	2	1
MaskedScale	28.4	7	8	8	2	2	0

Avg. score = macro-average over the eight models; Nonzero models = models with a nonzero score; 20/20 traj. = trajectories that passed all 20 evaluation stages.

6.2 Self-evolving effectiveness

To evaluate whether iterative optimization on the benchmark improves kernels, we compared the public-case and hidden-case scores of 38 operators. The hidden score is higher than the public score for 26 operators, with a mean score gain of 2.5 points and a mean baseline-speedup gain of 0.51×. Table 4 lists five representative operators. The improvement also appears in the theoretical speedup against the HAP hardware limit: for Gelu it rises from 0.290× to 0.366×, and for Sigmoid from 0.548× to 0.615×. This shows that iterative optimization moves kernels closer to the physical limit. Since only the public cases are visible during iteration, the higher hidden scores indicate that the optimization generalizes to unseen inputs.

6.3 HAP validity and attainability

Table 5 reports the number of public cases with measured and proxy baselines across the 910b2, 950pr, and 950dt metadata sets. The Unit-Peak Overlap Bound is instantiated identically for 910b2 and 950pr with platform-specific constants, and 950dt reuses the 950pr decomposition with a different HBM read bandwidth. Measured baselines are collected with warmup and repeated timing, and cases without a direct measurement use a capped proxy derived from the hardware limit.

Across all three hardware metadata sets, every public case satisfies $T_{\mathrm{HW}} \leq T_{\mathrm{baseline}}$, with no observed violation. The median ratio $T_{\mathrm{HW}}/T_{\mathrm{baseline}}$ is 0.100 for 910b2, 0.226 for 950pr, and 0.300 for 950dt; the 10th–90th percentile ranges are 0.100–0.431, 0.100–0.673, and 0.300–0.300, respectively. The ratio remains strictly below one for nearly all measured cases.

Each case records the dominant hardware bottleneck used in the HAP derivation. Across the 1,060 initial cases, memory transfer dominates in 63%, vector computation in 26%, and cube computation in 11%. This shows that the benchmark targets the NPU's real memory bottleneck and identifies memory transfer as the primary optimization target for NPU operators.

To verify that HAP rewards genuine kernel optimization, we compare the score with the platform-reported geometric-mean speedup against the hardware limit. We restrict to the 34 submissions that pass at least 80% of cases so that correctness is comparable. Among these submissions, the score correlates with the geometric-mean speedup against the baseline (Pearson $r = 0.78$) and against the HAP limit ($r = 0.71$). Figure 4 plots each submission's score against its proximity to the hardware limit: the dashed trend is clearly positive. The best submission attains 24.0% of the limit and the top-10 median attains 13.3%, which indicates that substantial headroom remains and the benchmark is far from saturated. A single degenerate entry that passes only 20/1060 cases yet reports an 81× baseline speedup and an 11.5× speedup against the limit is an audit signal that the correctness and coverage gates are designed to flag.

Table 4: Public-case and hidden-case scores after iterative optimization.

Operator	Public score	Hidden score	Public speedup	Hidden speedup
Gelu	66.8	82.2	0.68×	2.32×
Sigmoid	81.2	92.2	1.23×	3.17×
Exp	81.6	91.8	1.26×	3.02×
RmsNorm	64.2	77.4	0.60×	1.51×
Softmax	57.2	67.2	0.29×	0.98×

Table 5: Baseline and HAP anchor coverage for the initial suite.

Hardware	Cases	Measured	Proxy	Coverage	Mean baseline	Mean HAP
910b2	1,060	535	525	50.5%	248.2 μs	42.6 μs ×5.8
950pr	1,060	708	352	66.8%	167.5 μs	39.4 μs ×4.3
950dt	1,060	1,058	2	99.8%	67.5 μs	22.0 μs ×3.1

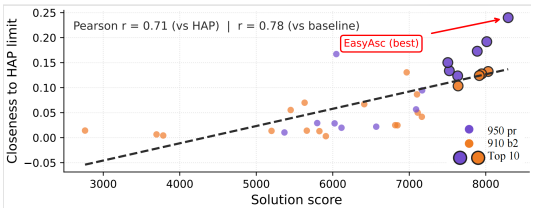

Figure 4: Each point is one of the 34 well-covered submissions (at least 80% of cases passed). The x-axis is the score; the y-axis is the geometric-mean speedup against the HAP hardware limit (higher means closer to the theoretical limit). Color denotes hardware, and larger markers highlight the top-10 solutions. The dashed least-squares line shows that higher scores correlate with approaching the limit (Pearson $r = 0.71$).

7 Limitations and Future Work

The current initial suite targets one NPU family and a specific software-stack version range. The two mechanisms generalize beyond this platform: SETG derives cases from operator computation structure and numerical error, while HAP's Unit-Peak Overlap Bound requires only operator compute and byte counts together with hardware bandwidth and throughput parameters, so both can be instantiated on other NPUs and accelerators. Precision coverage is centered on FP16, BF16, FP32, and integer paths, with low-precision formats planned as extensions. The analytical HAP limit is a hardware model, not a per-kernel optimal runtime, so cases that exceed the limit are treated as audit signals. Future work will extend kernel-DSL coverage and add lower-precision formats.

8 Conclusion

We presented a benchmark for AI-generated NPU kernels that combines a curated four-level operator suite, a self-evolving test-case methodology, and a hardware-attainable performance metric. The initial suite currently contains 53 operators and 1,060 cases, with baseline and HAP anchors available across multiple hardware metadata sets. The benchmark is designed to provide the NPU ecosystem with a reproducible, hardware-grounded evaluation scale and to evolve with new workloads, precisions, and silicon.

AI USE STATEMENT

This draft was prepared with assistance from an AI writing tool. All benchmark statistics were obtained from the public repository and the cited technical report. Authors reviewed and are responsible for all claims and numbers.

REPRODUCIBILITY STATEMENT

The benchmark suite, test-case files, golden implementations, baseline metadata, and evaluation scripts are available in the public repository described in the CANN Bench technical report (Gao et al., 2026; CANN Bench Contributors, 2026). The analytical HAP model and representative operator derivations are included in the repository documentation. Hardware constants used to compute the hardware limit should be taken from official hardware documentation; the public metadata already contains the resulting per-case anchors.

REFERENCES

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels, 2025. URL https://arxiv.org/abs/2507.11948.

CANN Bench Contributors. Cann bench technical report. https://gitcode.com/cann/cann-bench/blob/master/docs/technical-report.pdf, 2026. Repository technical report.

Xinzi Cao, Jianyang Zhai, Pengfei Li, Zhiheng Hu, Cen Yan, Bingxu Mu, et al. Ascendkernelgen: A systematic study of llm-based kernel generation for neural processing units, 2026. URL https://arxiv.org/abs/2601.07160.

Xin Cheng, Rui Tian, Wangding Zeng, Damai Dai, Qinyu Chen, Bingxuan Wang, Zhenda Xie, et al. Conditional memory via scalable lookup: A new axis of sparsity for large language models, 2026. DeepSeek Engram.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, et al. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation, 2026. URL https://arxiv.org/abs/2602.24286.

Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, et al. Microscaling data formats for deep learning, 2023. URL https://arxiv.org/abs/2310.10537.

Xue-Jian Gao, Deng Pan, Yueming Su, Jiasheng Li, Bin Du, Fengming Zhu, Chengdi Ma, Junyi Fan, Qichen Liao, Chengqiu Hu, et al. Cann bench: Benchmarking agent generated kernels against real npu and algorithmic limits, 2026. URL https://arxiv.org/abs/2607.20518.

Ping Guo, Chenyu Zhu, Siyuan Chen, Fei Liu, Xi Lin, Zhichao Lu, et al. Evoengineer: Mastering automated cuda kernel code evolution with large language models, 2025. URL https://arxiv.org/abs/2510.03760.

Hojae Han, Seung-won Hwang, Rajhans Samdani, and Yuxiong He. Convcodeworld: Benchmarking conversational code generation in reproducible feedback environments. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2502.19852.

Huawei Ascend. msprof profiling tool: Ascend cann auxiliary development toolkit documentation. https://www.hiascend.com/document, 2023. Huawei Ascend documentation page.

Jaber Jaber and Osama Jaber. Autokernel: Autonomous gpu kernel optimization via iterative agent-driven search, 2026. URL https://arxiv.org/abs/2603.21331.

Sarunas Kalade and Graham Schelle. Npueval: Optimizing npu kernels with llms and open source compilers, 2025. URL https://arxiv.org/abs/2507.14403.

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.

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

Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: A scalable and unified architecture for ubiquitous deep neural network computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 789–801. IEEE, 2021. doi: 10.1109/HPCA51647.2021.00071.

Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, et al. Sol-execbench: Speed-of-light benchmarking for real-world gpu kernels against hardware limits, 2026. URL https://arxiv.org/abs/2603.19173.

Jiayi Nie, Haoran Wu, Yao Lai, Zeyu Cao, Cheng Zhang, Binglei Lou, Erwei Wang, Jianyi Cheng, Timothy Jones, Robert Mullins, Rika Antonova, and Yiren Zhao. Kernelcraft: Benchmarking for agentic close-to-metal kernel generation on emerging hardware, 2026. URL https://arxiv.org/abs/2603.08721.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? In *Proceedings of the 42nd International Conference on Machine Learning*. PMLR, 2025. doi: 10.5555/3780338.3782232.

Ruizhong Qiu, Weiliang Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How efficient is llm-generated code? a rigorous & high-standard benchmark. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2406.06647.

Albert Tseng, Yu Tao, and Youngsuk Park. Training llms with mxfp4, 2025. URL https://arxiv.org/abs/2502.20586.

Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Chao Xu, Bin Ding, Ziqiong Liu, et al. Geak: Introducing triton kernel ai agent & evaluation benchmarks, 2025. URL https://arxiv.org/abs/2507.23194.

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, et al. Ascendcraft: Automatic ascend npu kernel generation via dsl-guided transcompilation, 2026. URL https://arxiv.org/abs/2601.22760.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi: 10.1145/1498765.1498785.

Jiehao Wu, Zixiao Huang, Wenhao Li, Chuyun Shen, Junjie Sheng, and Xiangfeng Wang. Ascendoptimizer: Episodic agent for ascend npu operator optimization, 2026. URL https://arxiv.org/abs/2603.23566.

Erkun Zhang, Pengxiang Xu, and Lu Lu. Efficient low-precision gemm on ascend npu: Hgemm's synergy of pipeline scheduling, tiling, and memory optimization. *Computers*, 15(1):39, 2026. doi: 10.3390/computers15010039.

Yujie Zheng, Zhuo Li, Shengtao Zhang, Hanjing Wang, Junjie Sheng, Jiaqian Wang, et al. Towards cold-start drafting and continual refining: A value-driven memory approach with application to npu kernel synthesis, 2026. URL https://arxiv.org/abs/2603.10846.

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?, 2026. URL https://arxiv.org/abs/2603.11935.