MEASURING THE CHECKER: MUTATION ANALYSIS FOR GPU-KERNEL BENCHMARK ORACLES

Anonymous authors
Paper under double-blind review

ABSTRACT

Benchmarks for LLM-generated GPU kernels judge correctness with a few random inputs and a loose floating-point tolerance, and their verdicts now rank models and reward reinforcement learning. They are patched by hand, with no way to measure whether a patch suffices. We introduce mutation analysis as an adequacy metric for kernel-benchmark oracles: deterministic rules inject faults into verified CUDA implementations of KernelBench problems, a fault counts only once a kill witness proves it detectable, and a protocol is scored by the fraction it detects. The official check misses **one in six** witnessed faults (16.9%), unevenly: 8.7% of arithmetic faults escape but 78.6% of precision faults do. Two mechanisms explain them, a tolerance blind band that grows with reduction size and an input distribution that never reaches the faulty path, and a measured ceiling on input aggressiveness, set by legitimate floating-point variance, bounds every remedy. The metric prices existing patches: KernelBench-Verified gains more from its tighter tolerance ($+4.5$) than from its four hidden input distributions ($+4.0$), and a published fuzzing recipe rejects *correct* kernels under the benchmark's own oracle. Set cover over the kill matrix reaches 98.0% detection with two inputs per problem and generalizes to held-out faults. At architecture scale blindness deepens, concentrating in long homogeneous pipelines, and two problems are unrefereeable: their official references violate the benchmark's tolerance against fp64.

1 INTRODUCTION

A benchmark's correctness checker used to be bookkeeping. For GPU-kernel generation it has become infrastructure that carries load: KernelBench-style verdicts (Ouyang et al., 2025) rank models publicly, gate which generated kernels reach production experiments, and, most consequentially, serve as the reward signal for reinforcement learning systems that write kernels (Baronio et al., 2025). A weak checker in this position fails silently. The model does not learn to write correct kernels; it learns to write kernels that *pass*, and the two diverge exactly where the checker is blind. The blindness is not hypothetical: on one KernelBench problem a kernel that returns all zeros passes every official trial, and no number of extra random seeds would catch it (§2).

The community has noticed. KernelBench-Verified (Zhang et al., 2026) documents generated kernels that hard-code their way past the official check, adds four hidden input distributions and a tighter tolerance, and reports headline speedups collapsing from $1.43\times$ to $0.88\times$. The Correctness Illusion (Sarkar, 2026) reaches the same verdict by seeding nine bugs by hand and proposing a fuzzing oracle; robust-kbench (Lange et al., 2025) hardens shapes and timing. Each of these efforts *patches* the checker. None can answer the question a maintainer actually faces: *does my patch cover the faults that matter, and what does it still miss?* A handful of hand-seeded bugs measures little; as §5 shows, a patch can look decisive on anecdotes while entire fault families remain unreachable *by construction*, no matter how many inputs the checker samples.

Software engineering solved this measurement problem fifty years ago. Mutation analysis (DeMillo et al., 1978; Jia & Harman, 2011) scores a test suite by the fraction of small injected faults it detects, turning "is my test suite good?" from opinion into measurement. Mutation faults are honest mistakes, not adversarial submissions: the score measures the blind band an ordinary bug can fall into, which lower-bounds what a policy optimizing against the checker can exploit (a check that accepts an all-zeros softmax will be gamed), but it does not measure gaming itself.

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

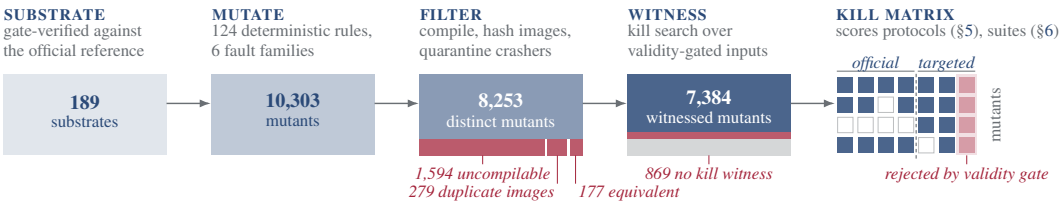

Figure 1: The measurement pipeline. Block height is proportional to mutant count; a red strip beneath a block is what that stage cut, split by cause and carried forward in grey so columns stay comparable. Deterministic rules mutate 189 gate-verified CUDA substrates into 10,303 mutants; filtering leaves 8,253 behaviourally distinct, and a kill-witness search over validity-gated inputs admits 7,384 into the scoring denominator. In the kill matrix (mutants $\times$ inputs, filled = detected), an input that rejects the *correct* kernel is discarded (red column) and no kill it records is credited.

The standard recipe does not transfer directly to GPU kernels: the oracle is a graded numerical comparison, aggressive inputs can reject *correct* implementations through legitimate floating-point variance, equivalent and oracle-blind mutants pollute the denominator, and per-mutant compilation makes naive matrices intractable (§3). We build the instrument and then read it. **KernelBench-M** pairs 189 KernelBench problems with gate-verified CUDA implementations, applies 124 deterministic mutation rules to them, compiles the resulting mutants at runtime, and records for every mutant and every candidate input whether the benchmark's own oracle rejects it: a kill matrix (Fig. 1). A fault enters the scoring denominator only once some validity-gated input has witnessed it, and a protocol's score is the fraction of witnessed faults it kills. We release the pool, witnesses, suites, and pipeline; our contribution is the instrument, what it reveals about the benchmarks the field stands on, and what it prescribes, in three parts.

- **The checker becomes measurable.** We adapt mutation analysis to graded numerical oracles: a measured *validity ceiling* below which harsher inputs still accept correct kernels, a *witnessed* denominator that admits only provably detectable faults, and runtime compilation that makes a 10,303-mutant kill matrix affordable. The official KernelBench check misses **one in six** witnessed faults (16.9%), and the misses concentrate exactly where real GPU bugs live: 78.6% of precision and 27.8% of synchronization faults escape, against 8.7% of textbook arithmetic mutations (Table 2). Two mechanisms explain them, *tolerance vacuity* and *input-domain evasion*, and the validity ceiling bounds every remedy (Fig. 2); across 48 whole architectures the blindness deepens rather than lifts, concentrating in long homogeneous pipelines (§4).
- **Patching blind fails in both directions** (§5). Scored on the same kill matrix, KernelBench-Verified gains more from its tighter tolerance (+4.5) than from its four hidden input distributions (+4.0), a split its authors state they cannot compute, and its constant scalings leave shape- and indexing-fault families unreachable *by construction*: 643 mutants escape every existing baseline. In the other direction, a published fuzzing recipe crosses the validity ceiling and rejects *correct* kernels 107 times under the benchmark's own oracle. Two level-3 problems prove unrefereeable altogether: their official references violate the benchmark's tolerance against fp64.
- **Measurement turns into better checkers** (§6). Set cover over the kill matrix yields two-input suites that detect 98.0% of witnessed faults (94.8% on held-out faults) against 83.1% for the official five: *chosen* testing, not more of it. The measurement's abstraction transfers, too: handing an LLM test generator the fault *taxonomy* nearly triples a fuzz baseline's kill rate (Appendix C).

2 The Checker Under Audit

The official protocol. A KernelBench problem supplies a PyTorch reference f and an input generator `get_inputs()`; a submission $\hat{f}$ passes if $\texttt{allclose}(f(x), \hat{f}(x); \texttt{atol}, \texttt{rtol})$ holds on a few draws, five in the official harness, at $\texttt{atol}=\texttt{rtol}=10^{-2}$. The inputs are dense random tensors at fixed, aligned shapes, and the tolerance and trial count are the same for every problem, whatever it computes. The verdict this check returns is the number that ranks models and rewards policies, so it is the object we measure. Two faults it cannot see, each verifiable by hand without our pipeline, show what a measurement must contend with.

A zeros-output softmax passes. One KernelBench problem applies softmax across $d = 393{,}216$ elements. Under the official `torch.rand` inputs the reference output averages 2.5×10^{-6} per element, while the check accepts any output within $\texttt{atol} = 10^{-2}$, four thousand times the signal. A kernel that returns all zeros passes every official trial. The blind band is not an edge case; it widens systematically with reduction size (Fig. 2b).

No reseeding can help. The official inputs are drawn from $[0, 1)$. Every element is positive, so deleting a ReLU is the identity on the entire support; $\exp(x)$ cannot overflow for $x < 1$, so removing softmax's max-subtraction stabilizer is unobservable. Survival under such inputs is a property of the *distribution*, not of sampling luck; more random trials measure the same blindness more confidently.

What the two faults teach. They instantiate two mechanisms that recur throughout the paper: *tolerance vacuity*, where a graded oracle admits any error below $\texttt{atol} + \texttt{rtol}\,|f|$, so a fault's visibility depends on the output's magnitude rather than on the fault; and *input-domain evasion*, where the fault's code path lies outside the support of the input distribution and no number of trials enters it. Neither is cured by more tests, only by different ones; and the obvious different tests, larger or harsher inputs, meet a constraint on every remedy that §3 measures, the *validity ceiling*: past a certain aggressiveness, correct kernels fail too. A metric must score inputs and tolerance together and know which inputs it may use.

3 Mutation analysis for graded numerical oracles

Setting. We want to score a *protocol*, inputs plus tolerance, by the fraction of faults it detects: writing M_W for the mutants that carry a kill witness (defined below) and $\mathrm{kill}_P(m){=}1$ when protocol P rejects mutant m on some input it accepts as valid, the *witnessed mutation score* is $\mathrm{MS}(P) = |M_W|^{-1} \sum_{m \in M_W} \mathrm{kill}_P(m)$. Rejection always uses the benchmark's own oracle (§2) unless a protocol specifies its own tolerance. Three adaptations separate this domain from classical mutation testing and from test-augmentation work on Python benchmarks (Liu et al., 2023b).

Inputs are gated at a measured validity ceiling. The oracle is graded: detection depends not on behavioural difference but on the ratio of a fault's error to the *output magnitude*, so the largest undetectable relative error is 0.39 at $d = 64$ and 2.5×10^3 at $d = 393{,}216$ (Fig. 2b). The obvious remedy, harsher inputs, is bounded above: two *correct* fp32 kernels legitimately disagree through accumulation order (Goldberg, 1991; Shanmugavelu et al., 2024). On a $K{=}4096$ reduction, sign-mixed inputs scaled by 300 push a correct kernel past the tolerance, while larger same-sign inputs remain safe because the tolerance scales with the un-cancelled output. Every input generator needs a validity gate at this ceiling; prior work has none; §5 shows a published recipe paying the price. An input is *valid* iff the unmutated substrate, compiled like its mutants, passes the benchmark's own oracle on it; invalid inputs are discarded and credit no kill (Fig. 1).

The denominator is earned by a kill witness. Some mutants are equivalent (a coherent `blockIdx` permutation relabels independent work); some are real faults no numerical oracle can see (an out-of-bounds write into allocator slack). Unkillable rows deflate every protocol equally and inform about none, so a mutant enters the denominator only with a *kill witness*, a validity-gated input on which it fails; the rest are quarantined. Where we report survival over all behaviourally *distinct* mutants, including the not-yet-adjudicated, we flag it; that looser rate upper-bounds oracle blindness.

Compilation is made tractable. Compiling one mutant as a standard `torch` extension takes $\sim$200 s, over three GPU-weeks for ten thousand mutants; runtime compilation (NVRTC) cuts this to 84 ms, a $\sim$2,400$\times$ reduction that makes the kill matrix affordable.

Pipeline. Fig. 1 summarizes the system. *Substrates:* mutation needs source, and KernelBench's references bottom out in closed cuDNN/cuBLAS binaries, so for each problem we maintain a correct CUDA implementation used *only* as a mutation target. Substrates are LLM-authored and admitted by an automated gate that checks them against the official reference on every suite available at admission: the official `get_inputs()` distribution plus the two to four targeted suites written with the substrate. The gate rejects real errors (an L1-norm dividing by sum instead of mean, for instance); a suite added later that rejects an admitted substrate is discarded as invalid, and the substrate stands. The substrate is a mutation target, never an oracle: every kill is adjudicated against the benchmark's own PyTorch reference, so a residual substrate flaw cannot manufacture one.

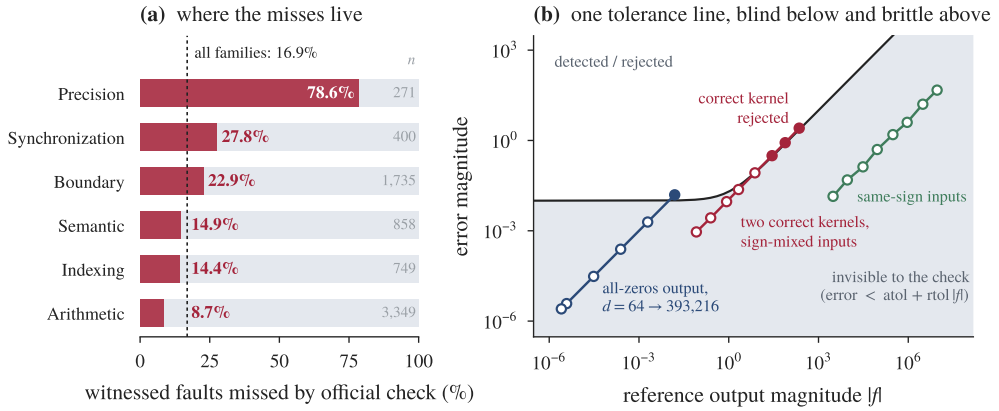

Figure 2: What the official check misses, and why. **(a)** Share of witnessed faults missed by the official protocol, by family (n at right). **(b)** The check accepts any error below $\texttt{atol} + \texttt{rtol}\,|f|$ (black line; grey region invisible to it; filled markers: rejected, open: accepted). *Vacuity:* "return all zeros" has error $|f|$ (blue); as the softmax reduction d grows it slides into the grey region and passes. *Validity ceiling:* fp32 accumulation noise between two *correct* kernels grows with input scale s; on sign-mixed inputs the output cancels and the noise climbs out of the region, rejecting a correct kernel at $s \geq 300$; on same-sign inputs it stays deep inside (green).

Mutants: 124 deterministic rules spanning six families (Table 2): classical arithmetic/relational replacements; GPU-specific operators in the lineage of Zhu & Zaidman (2020) and Bradbury et al. (2006) (barrier removal or weakening, ceil-to-floor grid division, bounds-guard deletion, fp16 accumulation, index-axis swaps); and LLM-mined fine-grained families (argmax tie-breaking, perturbed polynomial constants, shifted thresholds). *Filters:* NVRTC rejects 1,594 non-compiling mutants; hashing compiled images, the GPU analogue of Trivial Compiler Equivalence (Papadakis et al., 2015), removes 279 whose image duplicates another mutant's and 177 whose image equals the original's (host-only or no-op rewrites); tiny-input probes, a watchdog, and fork-level isolation quarantine crashers and hangs. *Witnesses:* the kill search sweeps, in order, the official distribution, the targeted suites written with each substrate, and the LLM-generated suites of §6 (which adjudicated a further 431 mutants); every suite passes the validity gate before any of its kills counts. Table 1 gives the pipeline in numbers.

4 How strong is the official check?

Headline. The official protocol, each problem's own `get_inputs()` at five seeds, detects 6,136 of the 7,384 witnessed faults: 83.1%. **One in six provably-detectable faults survives** (1,248). The rate is a population property, not sampling noise (bootstrapping over problems, the 90% band contracts from $[8, 29]\%$ at ten problems onto 16.9% at the full set), and it is heavily right-tailed: the median problem loses 10% of its witnessed faults while a tail of transposed-convolution and reduction problems loses 40–73% (Fig. 3). Where the misses live, by fault family, is the paper's central empirical fact (Fig. 2a, Table 2; per-operator detail in Appendix A).

Where the misses live. The gradient across families is stark. Textbook mutations, the operator swaps that dominate classical mutation testing, are caught at 91%: dense random inputs excite arithmetic everywhere, so arithmetic faults have nowhere to hide. The families that escape are precisely the ones real GPU bugs inhabit: *boundary* faults (22.9% missed) hide because official shapes are aligned and remainder blocks never execute (evasion); *synchronization* faults (27.8%) because small aligned workloads rarely lose the race (evasion); *precision* faults (78.6%) because the tolerance forgives them by construction (vacuity). Storing through an fp16 temporary, for instance, escapes 169 of 183 times (Appendix A). A checker validated on hand-seeded arithmetic bugs would look excellent and be blind where it matters.

Table 1: The mutation pipeline of Fig. 1 in numbers, at operator scale (levels 1–2) and architecture scale (level 3). "Distinct" counts mutants surviving compilation, compiled-image deduplication, and equivalence filtering; "witnessed" additionally requires a validity-gated input on which the mutant verifiably fails. The same 124 rules run at both scales (per-family split in Appendix B); 96 fire with a witness (Table 2). Substrates total 44,270 lines of CUDA across 360 device kernels, and the kill matrix comprises over 120,000 mutant–input evaluations. The level-3 witness search is far shallower (two to four targeted suites per problem, against the full matrix at operator scale), so its miss rate is a conservative lower bound and its distinct-mutant survival an upper bound.

	Operators (levels 1–2)	Architectures (level 3)
Problems (substrates)	189	48
Mutants generated	10,303	12,589
– non-compiling	1,594	38
– duplicate compiled image	279	155
– equivalent to original	177	35
Distinct mutants	8,253	12,361
of which witnessed	**7,384**	**8,519**
Witnessed, killed by the official inputs	6,136	7,043
Witnessed, missed by the official inputs	**1,248**	**1,476**
Miss rate (witnessed accounting)	16.9%	$\geq$ 17.3%
Survival rate (distinct accounting)	25.7%	**43.0%**

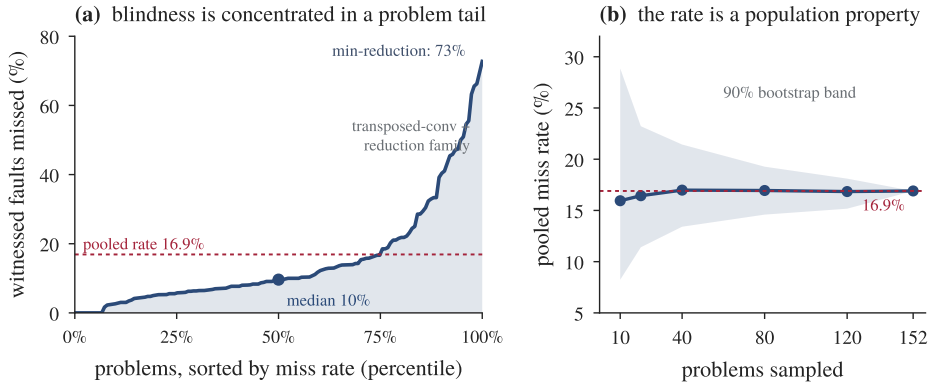

Figure 3: Population structure of the official check's blindness at operator scale. **(a)** Per-problem miss rate, sorted: half the problems lose under 10% of witnessed faults; a tail of transposed-convolution and reduction problems, the shapes vacuity predicts, loses 40–73%. **(b)** Pooled miss rate under bootstrap subsampling of problems: the 90% band contracts onto 16.9%, a property of the benchmark, not of our sample.

Mechanisms, quantified. The two mechanisms of §2 and the ceiling organize the survivors. *Vacuity scales with output magnitude:* softmax contributes 28 tolerance-blind survivors where structurally identical log-softmax contributes 9, because log-domain outputs are $O(10)$, collapsing the blind band of Fig. 2b. *Evasion cuts both ways:* 90 mutants are killed *only* by dense random inputs (under sparse or spiky inputs a wrong index usually reads another zero), so targeted inputs complement rather than replace the official distribution. *The ceiling caps the remedy:* the obvious fix, "test with harsher inputs," is unsound past the measured boundary of Fig. 2b, so our targeted suites use same-sign magnitudes. The empirically optimal suite is one dense-random input plus one or two structure-targeted ones, exactly what the optimizer of §6 discovers.

At architecture scale. Level-3 KernelBench problems are whole architectures; their substrates are multi-kernel pipelines (one `__global__` per layer, 2 to 140 kernels), so every mutant carries a layer tag and a question inaccessible at operator scale becomes measurable: *does an early fault reach the network output?* Architecture-level checking is weaker on every accounting (Table 1). Un-

Table 2: Fault taxonomy and where the official check fails: witnessed mutants per family and the share the official five-input protocol misses. *Rules* counts rules that produced at least one witnessed operator-scale mutant (full inventory of 124 in Appendix B). Family rows sum to 7,362; the remaining 22 witnessed mutants crashed the CUDA context under the official inputs (counted as a kill) and carry no family label.

Family	Rules	Witnessed	Missed	Example operators
Arithmetic	5	3,349	8.7%	$+\to-$, $*\to/$, *off-by-one constant*
Indexing	13	749	14.4%	*axis swap, stride confusion, transposed access*
Semantic	18	858	14.9%	*tie-breaking, fused-op reordering, flag flips*
Boundary	27	1,735	22.9%	*guard deletion, ceil→floor grid, tail drop*
Synchronization	6	400	27.8%	*barrier removal,* `__syncwarp` *weakening*
Precision	27	271	**78.6%**	*fp16 accumulators, no-stabilizer, fast-math*
All	96	7,384	16.9%	

der strict witnessed accounting 17.3% of faults escape the official inputs against 16.9% at operator scale, a floor, since the level-3 witness search is far shallower; the distinct-mutant upper bound is $1.7\times$ (43.0% versus 25.7%), inflated by the same shallow search, so it bounds architectural masking from above rather than measuring it. What realizes the masking is homogeneity, not depth (Fig. 6 in Appendix D.1): deep *homogeneous* pipelines are near-opaque to the official check (VGG-19 90%, SqueezeNet 90%, LSTM stacks 77–87%, a 17-layer MLP 82%), while networks dense in normalization and branching stay transparent at extreme depth (ResNet-18 at 51 kernels: 11%; SwinMLP at 71 kernels: 14%). The per-architecture median, 25.7%, matches the operator scale exactly, so the aggregate gap is carried entirely by the homogeneous tail; within networks, faults injected in the front half survive at 44–49% against 39% in the output quartile, a modest gap rather than a monotone gradient. The field's direction of travel, from single operators toward end-to-end generated models, is precisely the direction in which its oracles weaken.

5 Auditing hardened checkers

A metric that can score the official protocol can score its proposed replacements. We re-implement KernelBench-Verified (KBV) from its released source: four deterministic scalings of each problem's own inputs ($\times1$, $\times3$, $\times0.01$, $\times-1$; shapes never varied; integer tensors never scaled) plus an fp32 tolerance of 10^{-3}. We reconstruct the Correctness-Illusion-style fuzzer from its paper (log-uniform magnitudes spanning six decades at the official shapes; no code is released) and pair its inputs with the benchmark's own oracle, as a maintainer adopting the recipe into KernelBench would. The original pairs them with an fp64 CPU reference and per-dtype absolute tolerances, so the false rejections reported below are a property of the recipe's magnitudes under this oracle, not a claim about the original harness. Table 3 scores every protocol on a unified denominator of 8,215 witnessed mutants across 235 operator- and architecture-level problems: a mutant enters only if it is witnessed *and* has a recorded outcome under every audited column, so protocols are compared on identical mutants (Appendix D.2). Fig. 4 dissects the blind band into its miss-patterns.

Decomposing KBV. Of KBV's $+8.5$ points over the official protocol, $+4.0$ come from adding the four hidden distributions at the official tolerance and a further $+4.5$ from then tightening the tolerance to 10^{-3} on those inputs: **the tolerance change outweighs all four designed distributions combined**. The decomposition is sequential and the two factors can interact, so the split is specific to that order, and the claim rests only on which increment is larger. KBV's authors report the same asymmetry from the outside and state that their study "does not quantify how many failures were solely attributable to tighter tolerance versus distributional mismatches." The kill matrix computes exactly that split.

Why constant scalings plateau. All four KBV transforms rescale magnitude; none varies shape or structure. Remainder-block boundary faults and index-arithmetic faults are therefore unreachable *by construction*, however many scaled configurations are added, and these are exactly the families the official check already misses most often (Table 2). The ordering among the scalings is the

Table 3: Protocol audit on the unified denominator (8,215 witnessed mutants across 235 operator- and architecture-level problems; every protocol includes the official distribution). "Ours" adds per-problem inputs chosen by set cover on this same kill matrix, so its rows are *in-sample*; selecting on a dev half and scoring on held-out mutants (§6, operator scale) gives 94.8% at budget two against 82.9% for the official five on the same held-out half. The fuzz baseline's score counts kills recorded on inputs that reject correct kernels.

Protocol	Inputs / problem	Tolerance	Detected	Rejects correct kernels
Official KernelBench	5	10^{-2}	80.0%	0
KBV, hidden inputs only	4	10^{-2}	84.0%	0
Correctness-Illusion-style fuzz	3	10^{-2}	86.2%	**107**
KBV, inputs and tolerance	4	10^{-3}	88.5%	0
Ours, set cover (in-sample)	2	10^{-2}	**98.1%**	0
Ours, all targeted inputs	∼8	10^{-2}	99.9%	0

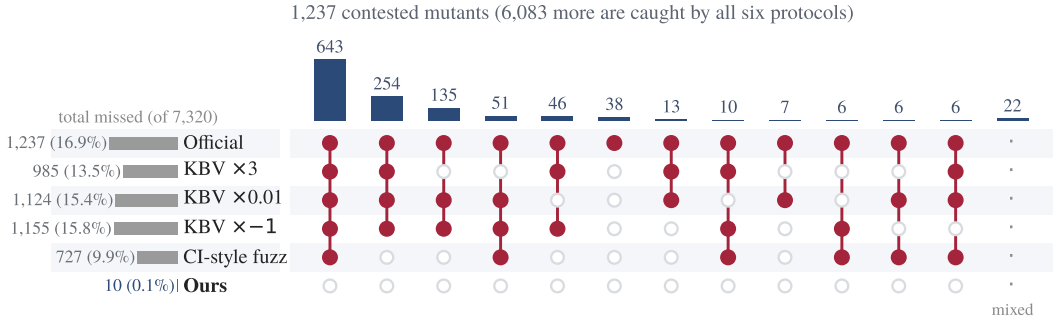

Figure 4: Anatomy of the blind band: the kill matrix collapsed to its distinct *miss-patterns* (UpSet form). Each column is one pattern of "which protocols miss this mutant" (filled = missed), its bar the number of operator-scale mutants with that pattern; left margins give each protocol's total misses. The band's core is the leftmost column: 643 mutants that every baseline misses and only our targeted suite detects. Each KBV scaling removes a different, partially overlapping slice (columns 2–5); the fuzzer removes more; our suite misses 10 of 7,320, all in the collapsed mixed tail (last column).

one vacuity predicts: $\times 3$ lifts outputs off the absolute-tolerance floor and recovers the most (985 misses against the official 1,237, Fig. 4); $\times 0.01$ pushes them toward it, and $\times -1$ changes only sign. Collapsing the kill matrix to its miss-patterns (Fig. 4) exposes the band's core: 643 operator-scale mutants that every baseline misses and only our targeted suite detects, a blind spot invisible to anecdote-based validation and obvious under a metric.

The fuzz baseline crosses the ceiling. The reconstructed fuzzer buys its 86.2% partly with invalid inputs: at the top of its magnitude range it rejected *correct* kernels 107 times, the Fig. 2b failure reached by a published recipe under the benchmark's oracle. That 86.2% is the recipe's *ungated* score, crediting kills recorded on the 107 rejecting inputs, exactly what §3 forbids for our own suites; under the validity gate those inputs and their kills are discarded. Detection bought with invalid inputs is not detection; a deployed benchmark would be rejecting honest submissions.

Two problems no oracle can referee. The two level-3 problems our admission gate never passed are unpassable in principle: on `48_Mamba2ReturnY` and `45_UNetSoftmax` the official fp32 reference itself violates the benchmark tolerance against its own fp64 evaluation, so no fp32 implementation, the reference included, can be adjudicated on them. Mamba2 exponentiates cumulative sums of unbounded random parameters, so outputs reach 10^{20} and 352 positions violate the tolerance; UNetSoftmax chains softmax into batch normalization, a variance amplifier, and deviates from fp64 by up to 0.74 at 9,379 positions. No prior audit noticed: KernelBench-Verified ships hidden tests for both, and its validity filter, which catches only NaN/Inf, passes a finite 10^{20}. Scores on these two problems are noise (Appendix D.3).

Table 4: Detection versus input budget on the witnessed level-1/2 pool. Held-out: suites selected on a 50% dev split of each problem's mutants and scored only on the other half.

	Optimized, budget b				Official
Scored on	b=1	b=2	b=3	b=5	5 random
Full pool	87.1%	**98.0%**	99.5%	100.0%	83.1%
Held-out mutants	84.5%	94.8%	96.4%	96.6%	82.9%

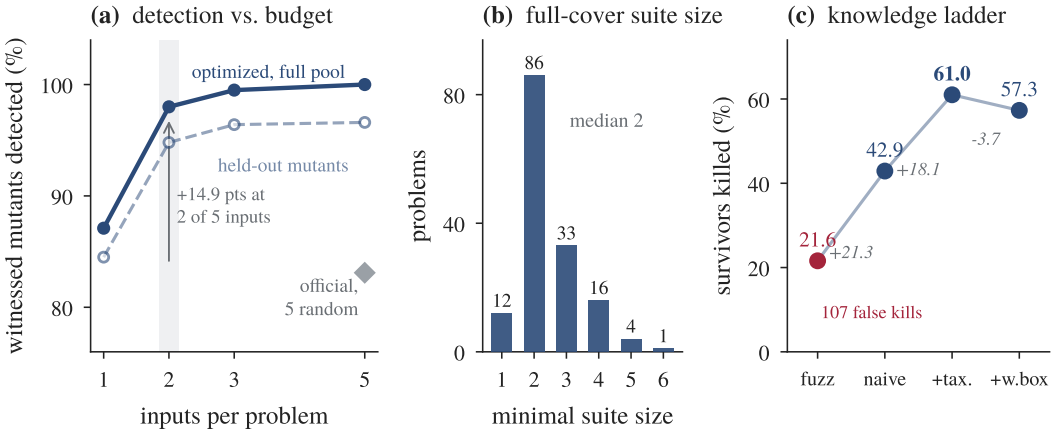

Figure 5: Synthesis. **(a)** Detection versus input budget under greedy set cover, in-sample (solid) and held-out (dashed). **(b)** Minimal full-coverage suite size per problem: median two, maximum six. **(c)** Knowledge ladder (Appendix C): official-input survivors killed as the generator is told more.

6 From measurement to synthesis

Suites as set cover. Once the kill matrix exists, suite construction is the classical covering problem (Harrold et al., 1993; Yoo & Harman, 2012). Greedy cover reaches full coverage of witnessed mutants with a median of **two** inputs per problem (Fig. 5b): one dense-random plus one targeted input suffice for 86 of the 152 problems with a non-trivial witnessed pool, and none needs more than six. The budget curve is steep (Fig. 5a; Table 4): at the official five-input budget an optimized suite detects every witnessed mutant, against 83.1%. The gain is *chosen* testing, not more: one chosen input already beats the official five (87.1% against 83.1%, Table 4), the optimizer's restatement of §2, since further draws from one distribution buy less than a single input from another.

Holdout validation. To rule out overfitting the selection to the measured pool, we split each problem's witnessed mutants 50/50 by id-hash, select on the dev half only, and score on the test half (Table 4, bottom row). Dev-selected suites detect 94.8% of held-out mutants at budget two, a ∼3-point generalization gap: the chosen inputs capture fault *families*, not memorized individuals, consistent with §4. Past two inputs the held-out curve is flat (96.4% at three, 96.6% at five): what remains is a matter of which inputs, not how many. The split is by mutant, not by source site; mutants at one site are often killed by the same input, so this rate is optimistic relative to a split by site.

Where the targeted inputs come from. Our targeted suites were written by an LLM given three artifacts of the measurement: the blind-spot taxonomy, a per-family attack playbook, and the ceiling of Fig. 2b. They killed 431 previously-unkilled mutants at a 94% suite-validity rate, and every rejected suite was a ceiling violation. Varying only what the generator is told (Fig. 5c) shows that the taxonomy nearly triples a fuzz baseline's kill rate on official-input survivors (21.6% → 61.0%), and that showing the generator the concrete surviving faults instead does not beat it: the white-box rung overfits its inputs to the sites it is shown, while family-level knowledge generalizes across the pool, so the measurement's abstraction is worth more than its raw instances.

7 Related work

Kernel benchmarks and their checkers. KernelBench (Ouyang et al., 2025) established the task and the randomized-`allclose` check we audit; TritonBench (Li et al., 2025) broadens the task surface. That the check is gameable is documented from both sides: Kevin (Baronio et al., 2025) reports reward hacking during RL against it, while KernelBench-Verified (Zhang et al., 2026), robust-kbench (Lange et al., 2025), and The Correctness Illusion (Sarkar, 2026) find bypasses and harden the protocol. All of these patch or exploit the oracle; none measures it. Our metric is the missing instrument, and §5 is what it reads on their patches.

Auditing benchmark oracles elsewhere. EvalPlus (Liu et al., 2023b) showed that HumanEval's tests pass wrong programs and that augmenting them reorders model rankings; EvalPerf (Liu et al., 2024) extends the audit to efficiency. Concurrently with us, program-variant test augmentation (Li et al., 2026) stresses SWE-bench Verified with semantically altered patches, and GateTruth (Bhadra, 2026) mutation-audits RTL benchmark testbenches, the closest cousin of our design. In every one of these domains a test passes or fails exactly: no fault hides below a tolerance, and two correct implementations never disagree, so neither tolerance vacuity nor a validity ceiling arises. Those two features force the validity gate and the witnessed denominator of §3.

Mutation testing and synthesis. Mutation analysis begins with DeMillo et al. (1978); Jia & Harman (2011) survey it, Offutt et al. (1996) ground operator selection, and Papadakis et al. (2015) give the equivalent-mutant filter whose CUBIN-hash analogue we use. On GPU code, MUTGPU (Zhu & Zaidman, 2020) and CLTestCheck (Peng & Rajan, 2019) score developer suites on hand-written kernels, Bradbury et al. (2006) pioneered concurrency operators, and GPUVerify (Betts et al., 2012) and test amplification (Leung et al., 2012) offer verification-flavoured alternatives; none targets a benchmark oracle or gates input validity. Goldberg (1991) and Shanmugavelu et al. (2024) characterize the floating-point non-associativity behind our ceiling; we measure where it bites. Suite minimization as set cover goes back to Harrold et al. (1993); Yoo & Harman (2012), and mutation-guided test generation is deployed industrially (Harman et al., 2025). Compiler-testing lines generate *equivalent* variants to expose miscompilation (Le et al., 2014; Liu et al., 2023a), where we generate *inequivalent* variants to expose weak checkers. Appendix D.5 expands on this.

8 Limitations

Adequacy is relative to a fault model: our 124 rules over naive substrates cannot express faults in structures we do not generate (tensor-core paths, double-buffered pipelines, some warp-level idioms), nor multi-site interactions. The metric therefore licenses *comparative* and *existence* claims (suite A misses faults suite B catches; this protocol misses these witnessed faults) but cannot certify a passing kernel correct or detect an adversarial submission, and we never use it to. A first realism check, 60 LLM-written leaderboard-style kernels gated against the official reference, found three incorrect, and both runtime bugs fall inside the taxonomy (Appendix D.6). Substrates are LLM-authored and gate-verified rather than sampled from submissions; results are from one GPU generation (H100); multi-GPU and distributed faults are out of scope. The level-3 study covers 48 of 50 architectures (the other two are unrefereeable, §5) with a shallower witness search, which Table 1 treats as a one-sided bound. The oracle and its tolerance are the benchmark's own by design: an fp64 reference would measure a different checker from the one that ranks models; §5 isolates the tolerance on its own.

9 Conclusion

Benchmarks for generated GPU kernels have been patching their correctness checkers blind. Mutation analysis, adapted to graded numerical oracles, turns a checker's blindness to honest faults into a number, prices existing patches, and converts suite design into optimization that generalizes to held-out faults. The prescription for maintainers is concrete: one dense-random input, one or two structure-targeted inputs below the validity ceiling, and no tolerance that admits an all-zeros output. We release the pool, witnesses, suites, and pipeline as **KernelBench-M**, so that the next patch to a kernel benchmark can ship with its coverage measured rather than asserted.

REPRODUCIBILITY STATEMENT

Every number in the paper is computed from journals shipped with the KernelBench-M artifact: the 124 mutation rules (Appendix B), the gate-verified CUDA substrates, per-mutant kill records with the suite that witnesses each kill, the dev/test split of §6, and the pipeline scripts that regenerate the kill matrix, the protocol audit, and the set-cover suites from those records. Figures are produced by the scripts in the repository from the released JSON summaries. All experiments ran on a single NVIDIA H100 (80 GB, driver 535.161.08) with PyTorch 2.7.1 (CUDA 12.6 build) and NVRTC 12.6; the measurement campaign used approximately 30 GPU-hours. NVRTC compiles every mutant with `--gpu-architecture=compute_90` (`sm_90` when emitting SASS), `--std=c++17`, and `-default-device`, with no fast-math or FMA-contraction override, so contraction defaults match the `nvcc` path of `torch`'s extension builder; the unmutated substrate is compiled through the same path as its mutants, so any residual compiler difference affects original and mutant alike.

ETHICS STATEMENT

This work audits the correctness checkers of public benchmarks; it involves no human subjects, personal data, or deployed systems. Its findings make benchmark verdicts harder to game and are intended to reduce, not enable, reward hacking in kernel-generating models. The audited benchmarks and the patches we re-implement are cited and used under their published terms; we report their weaknesses with the reconstruction details needed to check our claims and note where an earlier version of our own audit was at fault (Appendix D.4).

THE USE OF LARGE LANGUAGE MODELS

LLMs are part of the measured system, not only a writing aid: substrates are LLM-authored CUDA implementations admitted by an automated gate (§3), the fine-grained mutation families were mined with LLM assistance from the rule brief in the artifact, the targeted input suites of §6 and the knowledge-ladder rungs of Appendix C are LLM-generated under the prompts reproduced there, and the realism probe of §8 uses LLM-written optimized kernels. Every LLM-authored artifact was produced with the OpenAI Codex CLI (model `gpt-5.6-sol`) through one non-interactive `codex exec` call per task with a 900 s timeout, sampled once and retried only on failure; reasoning effort was left at its default (`none`) except for eight substrate-repair runs at `high`. No LLM output entered the paper unverified: every substrate passed the admission gate, every generated suite passed the validity gate before it could kill a mutant, and all numbers derive from the released journals. LLMs were also used to polish the prose.

REFERENCES

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

Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: A verifier for GPU kernels. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 113–132, 2012.

Meet Bhadra. GateTruth: Auditing the rigor of RTL design benchmarks via mutation testing. *arXiv preprint arXiv:2608.12635*, 2026.

Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Mutation operators for concurrent Java (J2SE 5.0). In *Second Workshop on Mutation Analysis (Mutation)*, 2006.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

Mark Harman, Jillian Ritchey, Inna Harper, Shubho Sengupta, Ke Mao, Abhishek Gulati, Christopher Foster, and Hervé Robert. Mutation-guided LLM-based test generation at Meta. In *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion)*, pp. 180–191, 2025.

Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.

Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 216–226, 2014.

Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying GPU kernels by test amplification. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 383–394, 2012.

Chenglin Li, Yisen Xu, Zehao Wang, Shin Hwei Tan, and Tse-Hsun Chen. Probe to generate: Program variant-guided test augmentation for repository-level repair benchmarks. *arXiv preprint arXiv:2604.01518 (to appear, ASE)*, 2026.

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

Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. NNSmith: Generating diverse and valid test cases for deep learning compilers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 530–543, 2023a.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023b.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation. In *Conference on Language Modeling (COLM)*, 2024.

Dolores Miao, Ignacio Laguna, Giorgis Georgakoudis, Konstantinos Parasyris, and Cindy Rubio-González. MUPPET: Optimizing performance in OpenMP via mutation testing. In *Workshop on Programming Models and Applications for Multicores and Manycores (PMAM@PPoPP)*, pp. 22–31, 2024.

A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

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 (ICML)*, volume 267 of *PMLR*, pp. 47356–47415, 2025.

Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pp. 936–946, 2015.

Chao Peng and Ajitha Rajan. CLTestCheck: Measuring test effectiveness for GPU kernels. In *Fundamental Approaches to Software Engineering (FASE)*, volume 11424 of *LNCS*, pp. 315–331, 2019.

Dipankar Sarkar. The correctness illusion in LLM-generated GPU kernels. *arXiv preprint arXiv:2606.20128*, 2026.

Sanjif Shanmugavelu, Mathieu Taillefumier, Christopher Culver, Oscar R. Hernandez, Mark Coletti, and Ada Sedova. Impacts of floating-point non-associativity on reproducibility for HPC and deep learning applications. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 170–179, 2024.

Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

Yunxiang Zhang, Ping Yu, Jianyu Wang, Max (Xiangjun) Fan, Julian Reed, Azalia Mirhoseini, and Will Su. KernelBench-Verified: Do LLM-generated kernels actually beat PyTorch? *arXiv preprint arXiv:2607.16241*, 2026.

Qianqian Zhu and Andy Zaidman. Massively parallel, highly efficient, but what about the test suite quality? applying mutation testing to GPU programs. In *IEEE International Conference on Software Testing, Validation and Verification (ICST)*, pp. 209–219, 2020.

A Operator inventory: where the misses come from

Table 5 lists the ten mutation operators contributing the most officially-missed faults at operator scale. The two regimes visible in Table 2 recur at operator granularity: high-volume arithmetic operators are almost always caught (6–10% missed), while low-volume implementation-level operators are missed at wholesale rates: storing through an fp16 temporary escapes the tolerance 92% of the time, and synchronization or boundary weakenings escape at 20–29% because the official aligned shapes never exercise the code they break.

Table 5: Top mutation operators by officially-missed faults (operator scale, witnessed accounting). "Example site" shows a representative mutated source location.

Operator	Family	Missed / witnessed	Rate	Example site
`store-fp16`	precision	169 / 183	92%	`out[i] = (half)acc`
`mul2plus`	arithmetic	99 / 990	10%	`delta * delta`
`loop-start-1`	boundary	82 / 417	20%	`for (int i = 0;`
`mul2div`	arithmetic	78 / 968	8%	`value * norm_weight[c]`
`loop-bound-minus1`	boundary	77 / 384	20%	`i < hidden_size; i++`
`plus2minus`	arithmetic	75 / 1,173	6%	`variance + 1.0e-5`
`sync-remove`	sync	58 / 198	29%	`__syncthreads();`
`sync2syncwarp`	sync	49 / 190	26%	`__syncthreads();`
`const0to1`	semantic	44 / 151	29%	`= 0.0f;`
`last-row-reduction-skip`	boundary	41 / 164	25%	`for (int j = 0; j < n;`

B Mutation-rule inventory

Tables 6–11 list every rule the released mutator applies, one table per fault family, with the one-line description recorded in the rule definition. Each rule is a deterministic rewrite (a regular-expression pattern and a replacement template) applied to the device source; every match site yields one mutant. Rules were added in three waves: the classical and GPU-specific operators of the base mutator, a set mined from survivor analysis, and a fine-grained set targeting numerically subtle sites; the rule brief that guided mining ships with the artifact. Four pairs of rules share a trigger pattern but apply different replacements (`lt2le`/`lt2gt`, `mul2div`/`mul2plus`, `acc-overwrite`/`acc-minus`, `sync-remove`/`sync2syncwarp`); they are distinct operators, and the family counts (5, 14, 22, 39, 14, 30) sum to the 124 used throughout.

Survival mechanisms. Where a rule's description explains why its mutants survive the official inputs, it cites one of four numbered mechanisms. They are the per-rule form of the regularities analysed in §4, and each names the kind of input that kills. *Mechanism 1 (tolerance absorption)*: the fault's error is real on the official inputs but smaller than the tolerance that the output magnitude admits, i.e. tolerance vacuity (§2, C1); a high-leverage value that enlarges the error relative to the output kills it. *Mechanism 2 (input-domain evasion)*: the mutated and original code take the same path on the entire support of the official distribution (e.g. positive-only inputs never enter a negative branch), the "no reseeding can help" case of §2; an input outside that support kills it. *Mechanism 3 (shape-alignment evasion)*: the fault lives in tail or guard logic that only fires when a size is not a multiple of the block or tile, and the official shapes are aligned, the blind spot that constant scalings cannot reach (§5); a misaligned shape kills it. *Mechanism 4 (benign race)*: a removed or weakened barrier produces a race whose outcomes are numerically indistinguishable under uniform random inputs, so the race is absorbed by the tolerance; inputs whose per-lane statistics differ sharply make the racing values diverge and kill it.

Table 6: Mutation rules: Arithmetic (5 rules).

Rule	Description
`minus2plus`	AOR: - → +
`mul2div`	AOR: * → /
`mul2plus`	AOR: * → +
`negate-store`	UOI: negate stored expression
`plus2minus`	AOR: + → -

Table 7: Mutation rules: Indexing (14 rules).

Rule	Description
`a-row-stride-n`	use N rather than K for one matrix-A row stride
`argmax-final-row-alias`	Mechanism 2 (input-domain evasion): it differs only when the unique maximum is in the final row; a final-row spike kills it.
`argmax-final-row-value-alias`	Mechanism 2 (input-domain evasion): only the last candidate is misindexed and is rarely maximal; a final-row spike kills it.
`b-row-stride-k`	use K rather than N for one matrix-B row stride
`bdim2gdim`	blockDim → gridDim in index math
`bidx2bidy`	wrong block axis (single site, unlike consistent-swap)
`flatten-height-use-width`	substitute width for one height stride in a flattened index
`grid-x-use-block-y`	use the other block extent in a linear global index
`output-position-height-first`	decode a flattened x coordinate with the wrong extent
`reduction-partner-half-offset`	read the wrong partner within one tree-reduction stage
`row-store-stride-k`	use the reduction dimension as one output row stride
`smem-index-off1`	shared-memory partner index off by one
`tidx2tidy`	wrong thread axis
`warpsize-16`	reduction start half → quarter

Table 8: Mutation rules: Semantic (22 rules).

Rule	Description
`acc-minus`	accumulate with $-=$ instead of $+=$
`acc-overwrite`	accumulate → overwrite (keeps last term only)
`argmax-nan-policy-drop`	drop the NaN-propagation branch: differs only on NaN inputs
`argmax-tie-last`	argmax keeps the LAST maximum instead of the first: differs only when duplicate values exist (measure zero under rand)
`argmax-tie-nan-form`	argmax with NaN handling: ties now keep the LAST index; only duplicate-valued inputs differ (measure zero under rand)

Rule	Description
const0to1	init constant $0 \rightarrow 1$
elu-positive-cutoff-nudge	Mechanism 2 (input-domain evasion): positive originals take the same branch; tiny negative values kill it.
fabs-simple-remove	absolute value is an identity on the original positive corpus
fmax2fmin	max $\rightarrow$ min
fmin2fmax	min $\rightarrow$ max
lower-negunit-clamp-remove	drop a lower saturation bound
nested-unit-clamp-drop-upper	retain the lower clamp but remove the upper saturation
reduce-fmax-tie-order	fmaxf replaced by a strict comparison: identical except for NaN operands, where fmaxf ignores NaN but $>$ propagates the old value
reduce-fmin-tie-order	fminf replaced by strict comparison: NaN semantics differ
relu-fmax-zero-first-remove	remove commuted ReLU wrapper; positive inputs conceal the fault
relu-fmax-zero-remove	remove a simple ReLU wrapper; differs only for negative values
relu-threshold-shift	activation threshold nudged off zero: only tiny values differ
selu-positive-cutoff-nudge	Mechanism 2 (input-domain evasion): ordinary positive values avoid the moved cutoff; tiny negative values in [-1e-6,0] kill it.
sentinel-zero	reduction sentinel -inf $\rightarrow$ 0 (breaks all-negative rows)
softsign-positive-fabs-elide	Mechanism 2 (input-domain evasion): fabs is identical for positive originals; any negative input kills it.
upper-unit-clamp-first-remove	drop a commuted unit upper clamp
upper-unit-clamp-remove	drop a unit upper clamp, exposed by values above one

Table 9: Mutation rules: Boundary (39 rules).

Rule	Description
arg-reduction-last-row-drop	Mechanism 2 (input-domain evasion): the final row is almost never the unique random extremum; forcing the extremum into the final row kills it.
ceil2floor	ceil-div $\rightarrow$ floor-div: tail block never processed
ceil2floor-lit	(x + K-1)/K with literal K $\rightarrow$ floor
cumsum-last-output-drop	Mechanism 1 (tolerance absorption): only the last prefix output is omitted and its missing term is tiny relative to a long positive sum; a short scan or last-position spike kills it.
elementwise-guard-minus1	last element of the flat range never written: needs a spiked or misaligned tail to expose
flat-n-one-extra-lane	Mechanism 3 (shape-alignment evasion): n is block-aligned in the original elementwise case; a misaligned n kills it by exposing the one-past-end access.
ge2gt	early-return guard $\geq \rightarrow >$: one OOB lane slips through
grid-stride-step-double	grid-stride loop skips every other element
groupnorm-sqdev-tail-scalar-drop	Mechanism 1 (tolerance absorption): one of many variance contributions is omitted; a high-leverage final outlier kills it.
groupnorm-sum-tail-scalar-drop	Mechanism 1 (tolerance absorption): one scalar among a huge group is absent from the mean; a dominant final scalar kills it.
gt2ge	ROR: $> \rightarrow \geq$
guard-drop-if	remove single-line bounds guard entirely (silent OOB)
guard-drop-ternary	remove edge-tile ternary guard: unconditional load
height-bound-use-width	use width as the vertical edge bound
last-row-reduction-skip	skip only the final item of a simple reduction loop
last-row-reduction-skip-start1	omit the last item while retaining a separately seeded first item
le2lt	$\leq \rightarrow <$: off-by-one underrun

Rule	Description
line-count-one-extra-lane	Mechanism 3 (shape-alignment evasion): aligned line counts launch no extra lane; a non-multiple line count with guarded allocation kills it.
literal-loop-last-skip	omit the last tap of a fixed-size pooling or convolution loop
loop-bound-minus1	drop last loop iteration (last tile / last row missing)
loop-start-1	skip first loop iteration
lt2gt	ROR: $< \to >$: inverted guard/loop condition
lt2le	guard/loop bound $< \to \leq$: off-by-one overrun
matmul-a-last-k-drop	Mechanism 1 (tolerance absorption): one of hundreds of positive dot-product terms is lost ($<$1%); a last-K spike kills it.
matmul-a-transposed-last-k-drop	Mechanism 1 (tolerance absorption): only the last term of a long dot product is removed; a dominant last A term kills it.
matmul-b-last-k-drop	Mechanism 1 (tolerance absorption): one reduction term among hundreds is zeroed; a last-K spike in B kills it.
matrix-col-bound-use-m	guard matrix columns with the row dimension
matrix-row-bound-use-n	guard matrix rows with the column dimension
min-reduction-last-row-drop	Mechanism 2 (input-domain evasion): a random final row is rarely the minimum; placing the unique minimum there kills it.
output-elements-one-extra-lane	Mechanism 3 (shape-alignment evasion): an exactly aligned launch has no lane at output_elements; a misaligned output plus a CUDA memory checker kills it.
pool2d-last-tap-drop	Mechanism 1 (tolerance absorption): one of 121 comparable positive pooling terms is lost (~0.83%); a large value in the omitted tap kills it.
reduce-seed-first-element	max-reduction seeded with 0 instead of -inf: wrong only when every value in the row is negative
reduce-seed-first-element-min	min-reduction seeded with 0: wrong only for all-positive rows
reduce-shift2	tree reduction skips levels
return-guard-drop	remove early-return bounds guard (silent OOB)
strided-tail-drop	drop the final strided chunk of a reduction or output pass
tail-guard-tighten	reject exactly the final valid scalar in a guarded tail
tile-last-iteration-skip	omit only the final matrix tile
width-bound-use-height	use height as the horizontal edge bound

Table 10: Mutation rules: Synchronization (14 rules).

Rule	Description
batchnorm-local-stats-warp-barrier	Mechanism 4 (benign race): uniform random lane statistics are close; per-lane distributions with very different means kill it.
batchnorm-merge-warp-barrier	Mechanism 4 (benign race): similar Welford partials hide stale cross-warp merges; alternating lane chunks kill it.
batchnorm-output-warp-barrier	Mechanism 4 (benign race): statistics are already reduced before this mostly redundant barrier; forced warp skew kills it.
groupnorm-final-output-warp-barrier	Mechanism 4 (benign race): the preceding reduction barrier normally makes this nearly redundant; adversarial scheduling with divergent lanes kills it.
groupnorm-sqdev-store-warp-barrier	Mechanism 4 (benign race): uniform inputs give similar squared-deviation partials; a lane-localized outlier kills it.
groupnorm-sum-store-warp-barrier	Mechanism 4 (benign race): similar lane partials mask cross-warp publication races; lane-chunk contrast kills it.
matmul-after-compute-warp-barrier	Mechanism 4 (benign race): stale and next-tile random values are statistically similar; a tile-alternating high/low input kills it.
matmul-after-load-warp-barrier	Mechanism 4 (benign race): warp-local progress often sees similar positive tiles; lane/warp-skewed tile values kill it.
reduction-barrier-move-before-store	move the barrier before the shared-memory publication
shared-denom-use-local	consume the per-thread value instead of the reduced shared value
shared-sqrt-use-local	read the local accumulator in place of the synchronized reduction
sync-remove	remove barrier: shared-memory race

Rule	Description
sync2syncwarp	__syncthreads → __syncwarp: block-wide barrier weakened to warp scope
welford-count-merge-early	publish a merged count to the partner lane instead of the consumer

Table 11: Mutation rules: Precision (30 rules).

Rule	Description
acc-fp16	fp32 accumulator declared fp16 (paired by fixup below)
avgpool121-denom-perturb	Mechanism 1 (tolerance absorption): the relative scale error is below 0.01%; a much tighter tolerance kills it.
batchnorm-epsilon-small-bias	Mechanism 1 (tolerance absorption): variance dominates a 1% epsilon perturbation; sub-epsilon variance kills it.
clamp-bound-perturb	saturation bound off by 1e-3: only saturated inputs differ
erf-to-tanh-approx	exact erf GELU replaced by the tanh approximation (~1e-3 relative)
expf-fast	expf → fast-math __expf approximation
fdividef-fast	true division → fast approximate division
gelu-sqrt2pi-perturb	sqrt(2/pi) constant mistyped: relative error ~1e-3
gelu-tanh-const-perturb	GELU tanh-approximation cubic coefficient off in the 4th digit
groupnorm-epsilon-halved	Mechanism 1 (tolerance absorption): ordinary random variance dwarfs epsilon; nearly constant inputs with variance below 1e-5 kill it.
hardsigmoid-half-ulpish-perturb	Mechanism 1 (tolerance absorption): a 1e-4 offset is below atol; a sub-1e-4 tolerance kills it.
hardsigmoid-six-denom-perturb	Mechanism 1 (tolerance absorption): the slope error is ~0.017%; a tight oracle near the linear-region edge kills it.
hardsigmoid-slope-perturb	hardsigmoid offset off by 5e-4: hidden by atol at small outputs
invsqrt2-perturb	1/sqrt(2) mistyped in the erf-based GELU
kahan-compensation-drop	remove a simple Kahan compensation subtraction
literal-epsilon-remove	drop a literal small stabilizer from a simple expression
matmul-acc-store-fma-nudge	Mechanism 1 (tolerance absorption): a 0.01% store-scale error is hidden by rtol; a tighter relative oracle kills it.
rsqrt-epsilon-remove	remove the variance epsilon safeguard
selu-alpha-perturb	SELU alpha constant digit transposition
selu-alpha-small-perturb	Mechanism 2 (input-domain evasion): the alpha is unused for positive originals; negative SELU inputs kill it.
selu-scale-perturb	SELU scale constant digit transposition
selu-scale-small-perturb	Mechanism 1 (tolerance absorption): the global relative error is under 0.01%; a tighter relative tolerance kills it.
softmax-max-subtraction-fast-remove	remove stabilization from a fast exponential
softmax-max-subtraction-remove	remove max subtraction, causing overflow only at large magnitude
softplus-cutoff-shift	softplus linear-regime threshold moved: wrong inside [8,20]
softplus-cutoff-tiny-shift	Mechanism 2 (input-domain evasion): original values never approach the large branch cutoff; values in [19.99,20] kill it.
sqrt-epsilon-remove	remove epsilon under a square root
store-fp16	output store through fp16 roundtrip
variance-unbiased-count	use sample rather than population variance
welford-second-delta-reuse	replace Welford's corrected second delta with the first delta

C KNOWLEDGE LADDER: WHAT MUST A GENERATOR KNOW?

To isolate the metric's contribution to test generation, we fix a hard target, the 1,154 mutants that survive all official inputs across 30 problems, and vary only the generator's knowledge (Table 12). R0 draws random inputs; R1 receives a naive prompt; R2 receives the taxonomy, the per-family playbook, and the ceiling, i.e. the targeted-suite generation of §6; R3 additionally sees the concrete surviving mutants (rule and mutated site). R1 and R3 generate one suite file per problem per rung; R3 is scored on 1,092 survivors because the evaluator skips a problem whose mutants crashed the CUDA context on two attempts (its crash re-entry guard), which excludes 62 survivors. Generation settings are those of the LLM-usage statement.

Two readings matter. First, each increment of *measurement-derived* knowledge buys detection; the taxonomy nearly triples the fuzz baseline ($21.6\% \rightarrow 61.0\%$). Second, white-box access to the faults themselves does *not* beat the taxonomy: shown the concrete surviving mutants, the generator overfits its inputs to them, while family-level knowledge generalizes to the whole pool. The comparison is not budget-matched (R2 accumulated more suites per problem than the single-shot R3), so the safe reading is that white-box access did not close the budget gap, not that it hurts; an equal-budget comparison is left open. Either way, the measurement's abstraction is worth more than its raw instances, which makes these prompted rungs a credible floor for a learned generator rewarded by the kill rate computed here.

Table 12: The knowledge ladder: the same task (kill 1,154 official-input survivors) under increasing generator knowledge. Validity = share of generated suites that do not reject the correct kernel. R2 accumulated more suites per problem than the single-shot rungs, so its edge is an upper bound. †R3 is scored on 1,092 survivors (see text). ‡R0 emits raw inputs rather than suites; 107 of them rejected the correct kernel.

Rung	Generator knows	Killed / evaluated	Rate	Valid suites
R0 random fuzz	nothing	249 / 1,154	21.6%	–‡
R1 naive prompt	"tests may miss subtle bugs"	495 / 1,154	42.9%	100% (60/60)
R2 + taxonomy & ceiling	fault families, safe scales	704 / 1,154	**61.0%**	94%
R3 + white-box diffs†	the mutated source sites	626 / 1,092	57.3%	98% (55/56)

The briefs handed to the generator at R1 and R3 follow verbatim (the R2 rung reuses the targeted-suite generation of §6, whose brief ships with the artifact); the invocation prepends only the assigned problem name and the path of its substrate.

R1 brief.

````
# Task: write extra test inputs for a GPU kernel benchmark problem

The benchmark checks a candidate CUDA kernel against a PyTorch reference with
torch.allclose(atol=1e-2, rtol=1e-2) on a few input tensors. You suspect the
current inputs may miss subtle bugs. Write 2 ADDITIONAL test-input suites for
the assigned problem.

Deliverable: ladder/suites_R1_<problem>.py containing exactly:

```python
import torch

def suites():
    def s1(trial):
        g = torch.Generator().manual_seed(1000 + trial)
        # return the SAME list of CPU tensors get_inputs() returns
        # (same count, dtypes, and any shape constraints the kernel requires)
        return [ ... ]
    def s2(trial):
        ...
    return [("R1_a", 2, s1), ("R1_b", 2, s2)]
```

Read the substrate file you are pointed at to learn the input shapes/dtypes
(its T0_original suite reproduces the official get_inputs()). Builders must be
deterministic (seed from trial), return CPU tensors, and respect any shape
````

````
constraints baked into the kernel/launcher. Do not modify any other file.
````

R3 brief.

````
# Task: kill specific surviving mutants of a GPU kernel (white-box)

The assigned problem has known SURVIVING MUTANTS: buggy variants of a correct
CUDA kernel that the benchmark's current inputs cannot distinguish from the
correct kernel (checker: torch.allclose(atol=1e-2, rtol=1e-2) vs the PyTorch
reference). You get the mutant list (rule + mutated source site) in the info
file you are pointed at. Write 2 test-input suites that make these mutants
produce detectably different output.

How to think: read each mutated site and construct inputs whose correct output
is sensitive to that exact code path -- boundary sizes that exercise remainder
blocks for guard/bounds mutants, values with distinct magnitudes per position
for indexing mutants, inputs that stress accumulation for precision mutants,
non-uniform data for tie-breaking mutants.

HARD VALIDITY CONSTRAINT: the CORRECT kernel must still pass. Two correct fp32
implementations legitimately differ by accumulation order; measured on long
reductions, SIGNED inputs (randn*s) are safe to about s=100 and falsely flag
the correct kernel by s=300, while SAME-SIGN inputs stay safe past s=1000.
Prefer same-sign magnitudes, moderate spikes, structured patterns -- not huge
signed values.

Deliverable: ladder/suites_R3_<problem>.py, exactly this contract:

```python
import torch

def suites():
    def s1(trial):
        g = torch.Generator().manual_seed(3000 + trial)
        return [ ... ]  # same list of CPU tensors as get_inputs()
    def s2(trial):
        ...
    return [("R3_a", 2, s1), ("R3_b", 2, s2)]
```

Deterministic builders, CPU tensors, keep every shape constraint the kernel
requires (read the substrate file's T0_original). Do not modify other files.
````

D SUPPLEMENTARY RESULTS AND PIPELINE DETAIL

D.1 ARCHITECTURE-SCALE SURVIVAL

Fig. 6 shows the per-quartile and per-architecture survival that §4 summarizes.

D.2 ADJUDICATION AND THE AUDIT DENOMINATOR

Adjudication. Of the mutants surviving every input known at screening time, LLM-generated targeted inputs later produced witnesses for 431 (§6); the remaining never-witnessed mutants are equivalence or oracle-blindness candidates; they stay outside the witnessed denominator and are charged to no protocol.

What the unified denominator of §5 contains. It is not the union of the two witnessed pools of Table 1. A mutant enters only if it is witnessed *and* has a recorded outcome under every audited column, so the audit compares protocols on identical mutants. At operator scale this removes from the 7,384 the mutants no competitor recipe can execute (crash-quarantined mutants, credited analytically in the audit but absent from the kill matrix) and the mutants whose only witness came from a suite the validity gate later rejected. At architecture scale the competitor columns were not re-run on all 8,519 witnessed mutants: each architecture contributes the per-(rule, layer) stratified sample on which its columns were evaluated, and two architectures whose audit did not complete are excluded, which is why 235 rather than 237 problems appear. Fig. 4 restricts further to operator scale and to mutants carrying a family label, hence its 7,320.

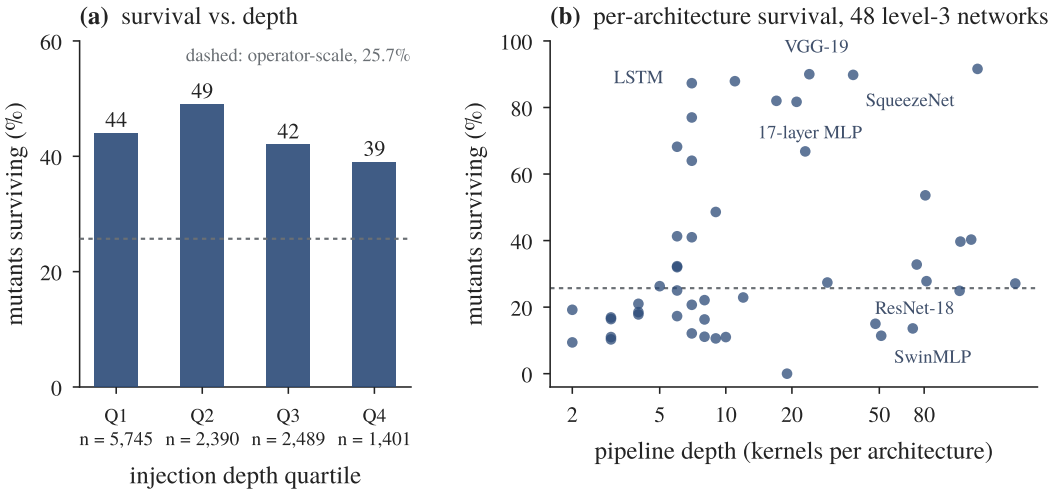

Figure 6: Architecture-scale results (distinct-mutant accounting). **(a)** Survival by injection-depth quartile across 48 architectures. **(b)** Per-architecture survival versus pipeline depth: deep homogeneous pipelines (VGG, LSTM, MLP chains) approach opacity; normalization- and branch-dense networks (ResNet-18, SwinMLP) stay checkable at any depth.

D.3 Two problems no oracle can referee

§5 gives the mechanism on each problem. Two further points. First, robust-kbench's filters never touch level 3, so its hardening cannot notice either problem. Second, the naive Mamba segment-sum is known upstream to be unstable (the reference implementation calls it `segsum_unstable`), but the instability had not been connected to the benchmark. Both problems are a concrete instance of what patching without measuring cannot see: every submission scored on them, correct or not, is adjudicated by a reference that would fail its own check.

D.4 A cautionary replication note

An early version of our own audit guessed $\mathcal{N}(0,1) \times 10^4$ for KBV's large-magnitude distribution and observed false rejections of correct kernels; the fault was our guess, not their protocol, whose published $\times 3$ cannot cross the ceiling. We keep the episode on record because it argues the thesis better than any experiment we designed: without a measured validity gate, an input designer, human or model, cannot tell when they have crossed it.

D.5 Extended related work

Kernel benchmarks and their checkers. KernelBench (Ouyang et al., 2025) established the task and the randomized-`allclose` check we audit; TritonBench (Li et al., 2025) and successors broaden the task surface. That the check is gameable is documented from both sides: Kevin (Baronio et al., 2025) reports reward hacking during RL against it, while KernelBench-Verified (Zhang et al., 2026) finds hard-coded bypasses and hardens the protocol, as do robust-kbench (Lange et al., 2025) and The Correctness Illusion (Sarkar, 2026). All of these patch or exploit the oracle; none measures it. Our metric is the missing instrument, and §5 is what it reads on their patches.

Auditing benchmark oracles elsewhere. EvalPlus (Liu et al., 2023b) showed HumanEval's tests pass wrong programs and that augmenting them reorders model rankings; EvalPerf (Liu et al., 2024) extends this to efficiency. Concurrently with us, program-variant test augmentation (Li et al., 2026) stresses SWE-bench Verified with semantically altered patches and GateTruth (Bhadra, 2026) mutation-audits RTL benchmark testbenches. None of these domains has a graded numerical oracle, so none confronts tolerance vacuity or a validity ceiling, the two features that make the kernel setting distinct (§3) and that our audit shows existing kernel-domain patches mishandle.

Mutation testing, including on GPU code. Mutation analysis begins with DeMillo et al. (1978); Jia & Harman (2011) survey it; Offutt et al. (1996) ground operator selection; Papadakis et al. (2015) give the equivalent-mutant filter whose CUBIN-hash analogue we use. On GPU code, MUTGPU (Zhu & Zaidman, 2020) defined CUDA-specific operators and CLTestCheck (Peng & Rajan, 2019) seeded faults to score OpenCL test suites; Bradbury et al. (2006) pioneered concurrency operators; GPUVerify (Betts et al., 2012) and test amplification (Leung et al., 2012) offer verification-flavoured alternatives. These score developer suites on hand-written kernels; none targets a benchmark oracle, and none gates input validity against floating-point variance.

Synthesis, compilers, numerics. Suite minimization as set cover goes back to Harrold et al. (1993); Yoo & Harman (2012); mutation-guided test generation is deployed industrially (Harman et al., 2025). Compiler-testing lines generate *equivalent* variants to expose miscompilation (EMI (Le et al., 2014), NNSmith (Liu et al., 2023a)), where we generate *inequivalent* variants to expose weak checkers. The floating-point non-associativity underlying our ceiling is characterized by Shanmugavelu et al. (2024) and Goldberg (1991); we measure where it bites and promote it to a hard constraint on test generation. Nearest to a mutation–performance crossover, MUPPET (Miao et al., 2024) mutates OpenMP directives to *search for* speedups, the same machinery aimed at the opposite problem. The combination needed here, mutation analysis grading the oracle of a kernel benchmark, did not previously exist.

D.6 Realism probe

For the probe of §8 we prompted an LLM to write leaderboard-style *optimized* kernels (tiling, `float4`, warp shuffles; no taxonomy shown) for 60 random problems and gated them against the official reference. Three were incorrect: a compile error, a misaligned-address crash from an unguarded vectorized load (our boundary/guard family), and a value error in a fused GEMM–GroupNorm (accumulation/semantic). The sample is small, but both runtime bugs fall inside the taxonomy. Beyond its three bugs, the probe exposed a harness blind spot that only realistic style triggers, the checker-measurement lesson applied to our own tooling: kernels qualified with `__launch_bounds__` defeated the naive `extern "C"` injection of the first harness. The released harness now matches the qualified form.