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

Mingzhe Du[1,2] **Anh Tuan Luu**[2,3] **Dong Huang**[1] **See-Kiong Ng**[1]
[1]National University of Singapore [2]Nanyang Technological University [3]CAIR, VinUniversity
{mingzhe, dhuang, seekiong}@nus.edu.sg, anhtuan.luu@ntu.edu.sg

ABSTRACT

Benchmarks for LLM-generated GPU kernels decide correctness with a few random inputs and a loose floating-point tolerance, and their verdicts now feed leaderboards and reinforcement-learning rewards. Recent work agrees these checkers are weak and patches them by hand—extra input distributions, fuzzing recipes, tighter tolerances—with no way to *measure* whether any patch suffices. We introduce mutation analysis as an adequacy metric for kernel-benchmark oracles: deterministic rules inject 10,303 compilable faults into verified CUDA implementations of 188 KernelBench problems, 7,384 of them with an independent kill witness; any test protocol is scored by the fraction it detects. The official check misses **one in six** witnessed faults (16.9%), deterministically, and the misses are skewed by family: 8.7% of arithmetic faults escape but 78.6% of precision faults do. The metric explains why (a tolerance blind band growing with reduction size; a measured ceiling on input aggressiveness set by legitimate floating-point variance), audits the strongest existing patch (KernelBench-Verified's gain splits into $+4.0$ points from hidden inputs and $+4.5$ from tighter tolerance, a split its authors could not compute), and exposes a published fuzzing recipe that rejects *correct* kernels 107 times. Optimizing suites over the kill matrix reaches 98.0% detection with two inputs per problem (94.8% held-out), and the measurement's fault taxonomy teaches a test generator more than the raw faults themselves. Across 48 whole architectures the blindness grows with scale, concentrating in deep homogeneous pipelines, and two problems prove unrefereeable: their official references violate the benchmark's own tolerance against fp64. We release everything as KernelBench-M.

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 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 that headline speedups collapse from $1.43\times$ to $0.88\times$ under the stronger protocol. 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 input shapes and timing methodology. Each of these efforts *patches* the checker. None of them can answer the question a benchmark maintainer actually faces: *does my patch cover the faults that matter, and what does it still miss?* Validating a protocol against ten hand-seeded bugs measures little; as we show in §5, a patch can look decisive on anecdotes while entire fault families remain unreachable *by construction*.

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. But the standard recipe does not transfer directly to GPU kernels: the oracle is a graded numerical comparison rather than an exact one,

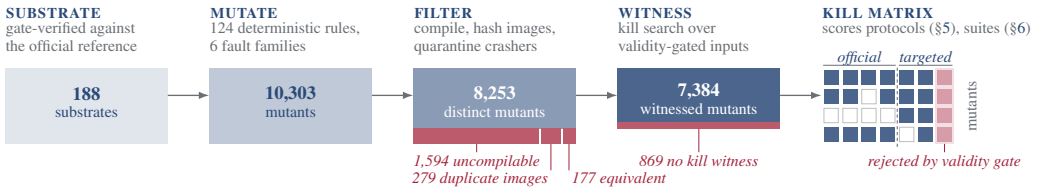

Figure 1: The measurement pipeline. Block height is proportional to mutant count. Deterministic rules mutate 188 gate-verified CUDA substrates into 10,303 mutants; compilation, compiled-image hashing, and crash quarantine remove 2,050 (the red strip cut from beneath each block, split by cause), and a kill-witness search over validity-gated inputs admits 7,384 into the scoring denominator, quarantining the 869 that no valid input kills. The witnessed kill matrix (mutants $\times$ inputs, filled = detected) records only inputs that pass the validity gate; an input that rejects the correct kernel (red column) is discarded. The matrix then scores any protocol, synthesizes minimal suites, and yields the fault taxonomy.

aggressive test inputs can reject *correct* implementations through legitimate floating-point variance, equivalent and oracle-blind mutants pollute the denominator, and per-mutant compilation cost makes naive matrices intractable (§3). Our central contribution is a mutation-analysis methodology that survives contact with these obstacles (Fig. 1)—and what the resulting metric reveals about the benchmarks the field is standing on.

Concretely: **(i)** We build the first at-scale measurement of kernel-benchmark oracle strength: 10,303 rule-generated faults across 188 verified CUDA substrates, 7,384 of them backed by kill witnesses so that only provably-detectable mutants enter any denominator (prior art validates against ≤ 10 hand-seeded bugs). The official KernelBench check misses 16.9% of witnessed faults—one in six—stably across scale and deterministically, with misses concentrated by family: 78.6% of precision faults and 27.8% of synchronization faults escape, against 8.7% of textbook arithmetic mutations (Table 2). **(ii)** We identify and quantify the domain's two governing mechanisms: *tolerance vacuity*, a blind band that grows with reduction size until an all-zeros output passes the check, and the *legitimate-variance ceiling*, a measured bound above which harsher inputs reject correct kernels (Fig. 2). **(iii)** We audit existing patches under their own published parameters (§5): KernelBench-Verified's improvement decomposes into $+4.0$ points from its four hidden distributions and $+4.5$ from its tighter tolerance—a decomposition its authors state they cannot compute—while its constant-scaling design leaves shape- and indexing-fault families unreachable; a reconstructed fuzzing baseline crosses the variance ceiling and falsely rejects correct kernels 107 times. **(iv)** We show measurement enables synthesis (§6): set cover over the kill matrix yields two-input suites at 98.0% detection (94.8% held-out) against 83.1% for the official five; and a knowledge-ladder experiment shows that giving a test generator the measurement-derived fault *taxonomy* nearly triples a fuzz baseline and outperforms giving it the concrete faults themselves. **(v)** We extend substrates to whole architectures (§7): across 48 networks and 12,361 distinct mutants, misses exceed operator level on every accounting (strict: 17.3% vs. 16.9%; upper bound: $1.7\times$), survival concentrates in deep homogeneous pipelines while normalization-dense networks stay checkable, and two further problems prove *unrefereeable*—their official references violate the benchmark's own tolerance against their fp64 selves. As generated kernels grow from operators into models, the checking problem gets harder, not easier.

2 Two faults the official check cannot see

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.

3 MUTATION ANALYSIS FOR GRADED NUMERICAL ORACLES

Setting. A benchmark problem supplies a PyTorch reference f and an input generator; a submission $\hat{f}$ passes if $\texttt{allclose}(f(x), \hat{f}(x); \texttt{atol}, \texttt{rtol})$ on a few draws. We want to score the *protocol*—inputs plus tolerance—by the fraction of faults it detects. Four obstacles separate this domain from classical mutation testing and from test-augmentation work on Python benchmarks (Liu et al., 2023b).

The oracle is graded (C1). Detection depends not on behavioural difference but on the ratio of a fault's error to the *output magnitude* entering the tolerance. We measure the consequence: at $d = 64$ the largest undetectable relative error is 0.39; at $d = 393{,}216$ it is 2.5×10^3 (Fig. 2b).

Valid inputs are bounded above (C2). 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 official tolerance—the input itself becomes invalid—while same-sign inputs of far larger magnitude remain safe, because the tolerance scales with the un-cancelled output (Fig. 2b). Every input-generation scheme for this domain needs a validity gate located by this ceiling; we found none in prior work that has one, and §5 shows a published recipe paying the price.

The denominator must be earned (C3). 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 landing in allocator slack). Scoring suites against unkillable rows deflates every protocol equally and informs about none. We therefore admit a mutant into the denominator only with a *kill witness*—a concrete, validity-gated input on which it verifiably fails—and quarantine the rest. Where we additionally report survival over all behaviourally *distinct* mutants (which includes the not-yet-adjudicated), we label it explicitly; that looser rate upper-bounds oracle blindness.

The Cost (C4). Compiling one mutant through the standard `torch` extension path takes $\sim$200 s; ten thousand mutants would need GPU-years. Runtime compilation (NVRTC) brings this to 84 ms—a $500\times$ 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; for each problem we maintain a correct CUDA implementation used *only* as a mutation target—the oracle remains the benchmark's own reference. Substrates are LLM-authored and admitted by an automated gate that checks them against the official reference on every suite (the gate rejects real errors, e.g. an L1-norm dividing by sum instead of mean). *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, `__syncthreads`→`__syncwarp`, 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 piecewise thresholds). *Filters:* NVRTC rejects 1,594 non-compiling mutants; hashing compiled images—the GPU analogue of Trivial Compiler Equivalence (Papadakis et al., 2015)—removes 456 duplicates and host-only no-ops; tiny-input probes, a watchdog, and fork-level isolation quarantine crashers and hangs. Table 1 gives the pipeline in numbers. *Scoring:* a protocol's score is the fraction of the 7,384 witnessed mutants it kills under the benchmark's own oracle (`allclose`, `atol=rtol=`10^{-2} unless the protocol specifies otherwise).

4 HOW STRONG IS THE OFFICIAL CHECK?

Headline. The official protocol—each problem's own `get_inputs()`, five seeds—detects 6,136 of the 7,384 witnessed faults: 83.1%. **One in six provably-detectable faults survives** (1,248). The pool behind these rates is summarized in Table 1: 124 rules fire 10,303 times across 44,270 lines of substrate CUDA, and over 120,000 mutant–input evaluations build the kill matrix. Bootstrap resampling over problems shows the pooled rate is a population property, not sampling noise: the 90% band contracts from $[8, 29]\%$ at ten problems onto 16.9% at the full set (Fig. 3b). The rate's structure matters as much as its value: the per-problem distribution is heavily right-tailed (Fig. 3a)—the median problem loses 10% of its witnessed faults while a tail dominated by transposed-convolution and reduction problems loses 40–73%—and where the misses live by fault family is the paper's central empirical fact (Fig. 2a, Table 2; per-operator detail in Appendix A).

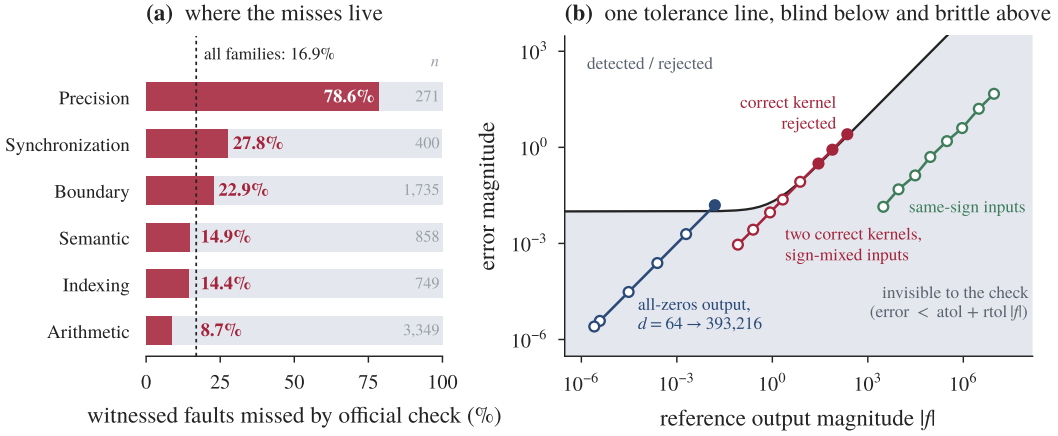

Figure 2: What the official check misses, and why. **(a)** Share of witnessed faults missed by the official protocol, by family (n at right): textbook arithmetic mutations are mostly caught; precision faults escape at $9\times$ the average rate. **(b)** The check accepts any error below $\texttt{atol} + \texttt{rtol}\,|f|$ (black line; grey region invisible to it; filled markers: the check fires, open: it accepts). *Vacuity:* the fault "return all zeros" has error $|f|$ (blue); as the softmax reduction d grows it slides into the grey region and passes. *Validity ceiling:* the 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 ($s \geq 300$); on same-sign inputs it stays deep inside (green).

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 124 rules split boundary 39, precision 30, semantic 22, synchronization 14, indexing 14, arithmetic 5; 96 fire with a witness (Table 2). Substrates total 44,270 lines of CUDA across 360 device kernels; at operator scale a problem carries 37 mutants at the median (199 at most) and 5 survivors (81 at most). The kill matrix comprises over 120,000 mutant–input evaluations, each mutant compiled in 84 ms by NVRTC versus 200 s through the torch extension path.

	Operators (levels 1–2)	Architectures (level 3)
Problems (substrates)	189	48
Mutation rules	124	124
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 mutants killed by the official inputs	6,136	7,043
Witnessed mutants missed by the official inputs	**1,248**	**1,476**
Miss rate	16.9%	17.3%

Where the misses live. Table 2 breaks the 16.9% down by fault family, and the gradient is stark. Textbook mutations—the operator swaps that dominate classical mutation testing—are caught at 91%: random dense 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; *synchronization* faults (27.8%) hide because small aligned workloads rarely lose the race; *precision* faults (78.6%) hide because the tolerance forgives them by construction. A checker validated on hand-seeded arithmetic bugs would look excellent and be blind where it matters.

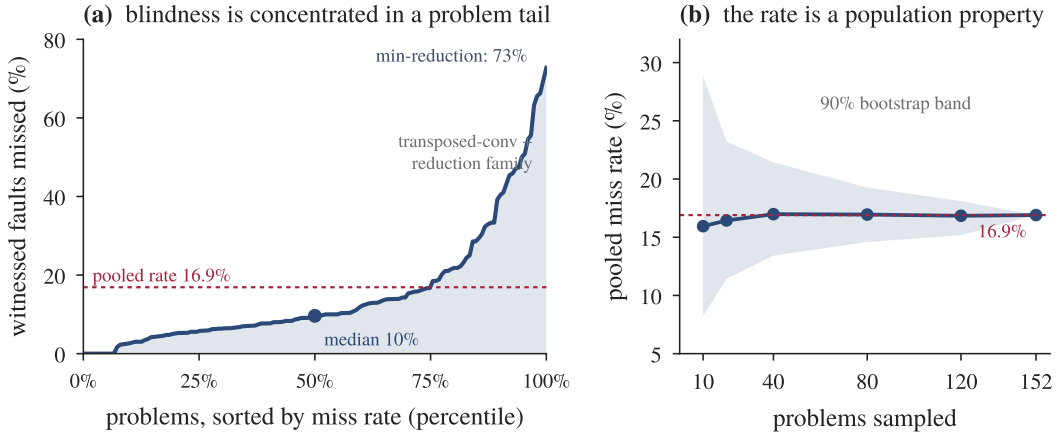

Figure 3: Population structure of the official check's blindness at operator scale. **(a)** Per-problem miss rate, sorted: half the problems lose under 10% of witnessed faults, while a tail of transposed-convolution and reduction problems—exactly the shapes tolerance vacuity predicts—lose 40–73%. **(b)**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

Table 2: Operator taxonomy and where the official check fails. Witnessed mutants per family and the share the official five-input protocol misses. The gradient is the paper's central empirical fact: textbook arithmetic mutations are mostly caught, while the implementation-level families real kernel bugs occupy—boundary, synchronization, precision—escape at 3× to 9× the rate.

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	7,362	16.9%	

Mechanisms, quantified. Three regularities organize the survivors. *Vacuity scales with output magnitude:* softmax contributes 28 tolerance-blind survivors where structurally identical log-softmax contributes 9—log-domain outputs are $O(10)$, collapsing the blind band of Fig. 2b. *The ceiling caps input aggressiveness:* the obvious fix, "test with harsher inputs," is unsound past the measured boundary of Fig. 2b; our targeted suites therefore use same-sign magnitudes. *Complementarity:* 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 empirically optimal suite shape is one dense-random input plus one or two structure-targeted ones, which is exactly what the optimizer of §6 discovers.

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 and stay outside the witnessed denominator—charged to no protocol.

5 Auditing Hardened Checkers

A metric that can score the official protocol can score its proposed replacements. We re-implement KernelBench-Verified from its released source: four deterministic scalings of each problem's own

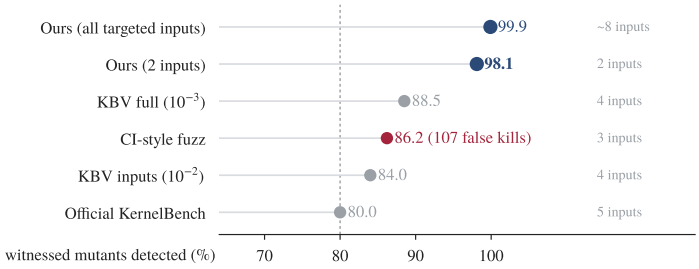

Figure 4: Protocol audit on the unified denominator (8,215 witnessed mutants across 235 operator- and architecture-level problems). "Ours" is the official dense-random input plus one per-problem input chosen by set cover (§6). Hardened protocols improve on the official check (dashed line) but plateau below optimized selection at half the input budget; the fuzz baseline's score comes with 107 false rejections of correct kernels. Fig. 5 dissects the blind band into its miss-patterns.

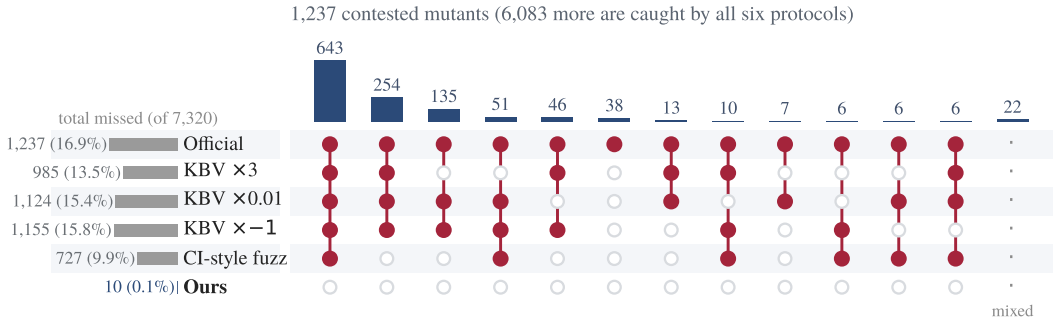

Figure 5: Anatomy of the blind band: the kill matrix collapsed to its distinct *miss-patterns* (UpSet form). Each column is one pattern of "which protocols miss this mutant" (filled = missed), its bar the number of operator-scale mutants with that pattern; left margins give each protocol's total misses (cf. Fig. 4). 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 in the residual mixed tail.

inputs ($\times 1$, $\times 3$, $\times 0.01$, $\times -1$; 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; no code is released). All protocols are scored on a unified denominator—8,215 witnessed mutants across 235 operator- and architecture-level problems—and every protocol includes the original distribution.

Decomposing KBV. Of KernelBench-Verified's $+8.5$ points over the official protocol, $+4.0$ come from the four hidden distributions and $+4.5$ from the tighter tolerance: **the tolerance change outweighs all four designed distributions combined**. Their paper reports the same asymmetry from the outside—hidden tests explain a 15% speedup drop at level 1 but only 3% at level 2—and states that it "does not quantify how many failures were solely attributable to tighter tolerance versus distributional mismatches." The kill matrix computes precisely this.

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*, independent of how many scaled configurations are added—exactly the families Table 2 shows the official check already misses most. This is the kind of 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 in our audit—the Fig. 2b

Table 3: 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%

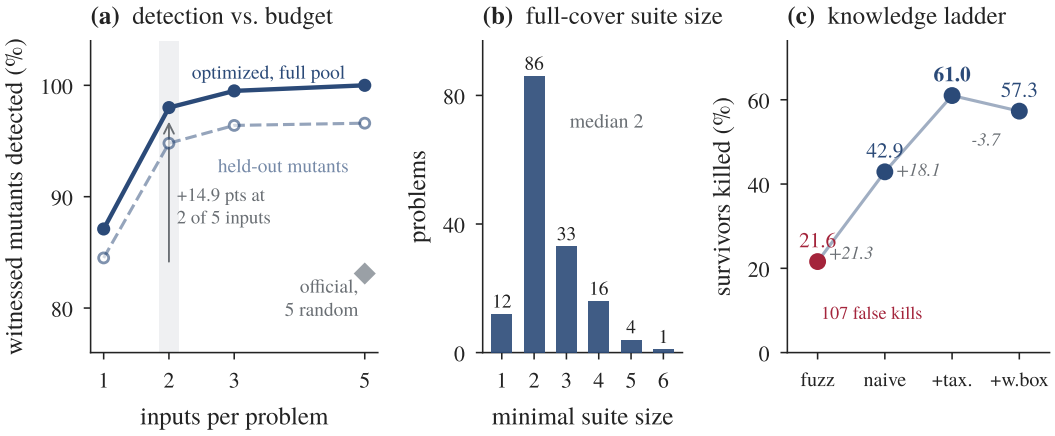

Figure 6: Synthesis results. **(a)** Detection versus input budget: greedy set cover on the kill matrix, scored on the full pool (solid) and on held-out mutants under dev-half selection (dashed). **(b)** Minimal full-coverage suite size per problem: median two inputs, maximum six. **(c)** Killing power of generated suites on 1,154 official-input survivors as generator knowledge grows (Table 4): the measurement-derived taxonomy beats both ignorance and white-box fault listings.

failure, committed by a published recipe. Detection bought with invalid inputs is not detection; a deployed benchmark would be rejecting honest submissions.

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—their 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.

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. 6b): one dense-random plus one targeted input suffice for 86 of the 152 problems with a non-trivial witnessed pool, and no problem needs more than six. The budget curve is steep (Table 3, Fig. 6a): at the official five-input budget an optimized suite detects every witnessed mutant, against the official 83.1%. The gain is *chosen* testing, not more.

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 3, bottom row). Dev-selected suites detect 94.8% of held-out mutants at budget two—an $\sim$3-point generalization gap— indicating the chosen inputs capture fault *families*, not memorized individuals, consistent with the mechanism analysis of §4.

Table 4: The knowledge ladder: identical task (kill 1,154 official-input survivors), increasing generator knowledge. Validity = share of generated suites that do not reject the correct kernel. The taxonomy rung accumulated more suites per problem than the single-shot rungs, so its edge over the naive prompt is an upper bound; the ordering is unaffected.

Rung	Generator knows	Survivors killed	Validity
R0 random fuzz	nothing	21.6%	107 false kills
R1 naive prompt	"tests may miss subtle bugs"	42.9%	60/60
R2 + taxonomy & ceiling	fault families, safe magnitudes	**61.0%**	94% (accum.)
R3 + white-box diffs	the mutated source sites	57.3%	55/56

Table 5: Operator scale versus architecture scale. Distinct = behaviourally distinct compilable mutants; survival = share not killed by the official inputs; witnessed = mutants with a validity-gated kill witness. The level-3 witness search is far shallower (2–4 targeted suites per problem versus the full matrix at operator scale), so its witnessed miss rate is a conservative lower bound and its distinct-survival an upper bound.

Scale	Problems	Distinct	Survival (distinct)	Missed (witnessed)
Operators (levels 1–2)	189	8,253	25.7%	16.9%
Architectures (level 3)	48	12,361	**43.0%**	$\geq$ 17.3%

What must a generator know? 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 the 6% of suites the gate rejected were precisely ceiling violations. To isolate how much of this is the metric's contribution, we fix a hard target—1,154 mutants that survive all official inputs, across 30 problems—and vary only the generator's knowledge (Table 4, Fig. 6c).

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 ten concrete mutants, the generator overfits its inputs to them, while family-level knowledge generalizes to the whole pool. The measurement's abstraction is worth more than its raw instances—which is what makes these prompted rungs a credible floor for a learned generator whose reward is the kill rate computed here.

7 Does depth hide faults?

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 5). Under strict witnessed accounting, 17.3% of architecture-level faults escape the official inputs against 16.9% at operator scale—and this is 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%. More telling than either aggregate is the structure. Per architecture (Fig. 7b), survival splits sharply: 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 architectures dense in normalization and branching stay comparatively transparent even at extreme depth (ResNet-18 at 51 kernels: 11%; SwinMLP at 71 kernels: 14%); the per-architecture median of 25.7% matches the operator scale exactly, so the aggregate gap is carried entirely by the homogeneous tail. Depth supplies the opportunity for masking; homogeneity—long chains without renormalization—realizes it. Within networks the same physics appears (Fig. 7a): faults injected in the front half survive at 44–49%, falling to 39% in the output quartile, a modest but consistent gradient ($n = 1{,}401$–$5{,}745$ per quartile). The direction of travel for the field—from single operators toward end-to-end generated models—is precisely the direction in which its oracles weaken.

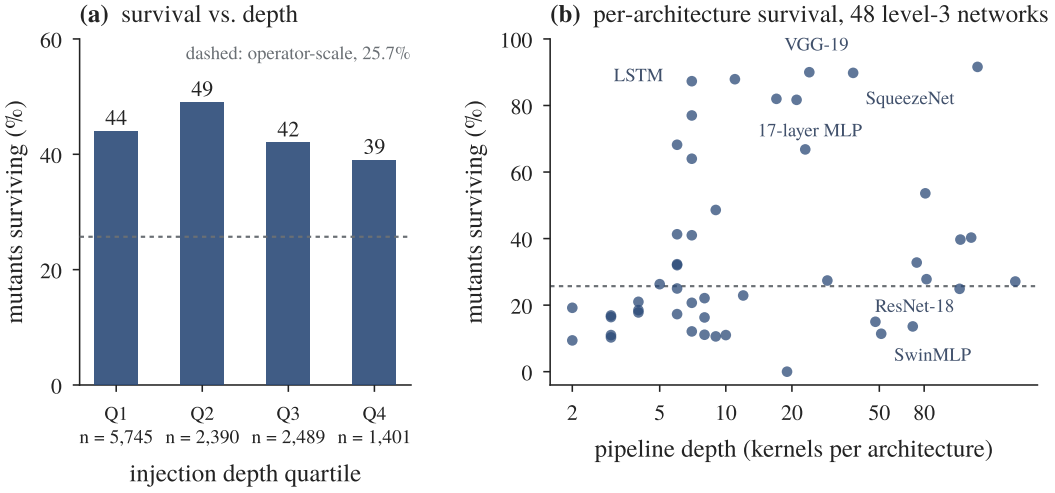

Figure 7: Architecture-scale results (distinct-mutant accounting). **(a)** Survival by injection-depth quartile across 48 architectures: faults with more network downstream survive more. **(b)** Per-architecture survival versus pipeline depth: deep homogeneous pipelines (VGG, LSTM, MLP chains) approach opacity, while normalization- and branch-dense networks (ResNet-18, SwinMLP) remain checkable at any depth.

Two problems no oracle can referee. The two level-3 problems our admission gate could never pass turn out to be unpassable in principle. For `48_Mamba2ReturnY`, whose reference exponentiates cumulative sums of unbounded random parameters, outputs reach 10^{20} and the official fp32 forward violates the benchmark tolerance against its own fp64 evaluation at 352 positions; for `45_UNetSoftmax`, whose blocks chain softmax into batch normalization—a variance amplifier—the official fp32 forward deviates from fp64 by up to 0.74 (9,379 violations), *farther than our rejected candidate sits from the reference*. No fp32 implementation, including the reference itself, can be adjudicated on these problems. No prior audit noticed: KernelBench-Verified ships hidden tests for both (its validity filter catches only NaN/Inf, and finite 10^{20} outputs pass); robust-kbench's filters never touch level 3; the instability of the naive Mamba segment-sum is acknowledged upstream (the reference implementation names it `segsum_unstable`) but had not been connected to the benchmark. Scores reported on these two problems are noise—a concrete instance of what patching without measuring cannot see.

8 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 mechanisms 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.

9 Limitations

Adequacy is relative to a fault model. Our 124 rules over naive substrates cannot express faults living in structures we do not generate— tensor-core paths, double-buffered pipelines, some warp-level idioms—nor multi-site interactions. The metric licenses *comparative* claims (suite A misses faults suite B catches) and *existence* claims (this protocol misses these witnessed faults); it cannot certify a passing kernel correct, and we never use it to.

Realism probe. As a first direct check on the fault model, 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—squarely in our boundary/guard family; and a value error in a fused GEMM–GroupNorm, matching our accumulation/semantic families. The bug sample is small, but both runtime bugs fall inside the taxonomy. The exercise also exposed a harness blind spot that only realistic style triggers (`__launch_bounds__`-qualified kernels defeating naive `extern "C"` injection)—the checker-measurement lesson applied to our own tooling.

Scope. Substrates are LLM-authored and gate-verified rather than sampled from submissions; results are from one GPU generation (H100). The level-3 study covers 48 of 50 architectures—the remaining two are excluded on proof that no fp32 implementation can be adjudicated on them (§7)—and its witness search is shallower than at operator scale, which Table 5 treats as a one-sided bound. The tolerance is fixed at the benchmark's own 10^{-2} by design, with §5 isolating the effect of changing it.

10 Conclusion

Benchmarks for generated GPU kernels have been patching their correctness checkers blind. Mutation analysis, adapted to graded numerical oracles—a witnessed denominator, a measured validity ceiling, tractable compilation— turns checker quality into a number, explains the number through two mechanisms and a six-family taxonomy, prices existing patches, and converts suite design into optimization that generalizes to held-out faults. The measurement also yields its most useful by-product for what comes next: a fault taxonomy that teaches a test generator more than the faults themselves. 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.

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 (§5).

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 and the knowledge-ladder rungs of §6 are LLM-generated under the prompts reproduced in Appendix C, and the realism probe of §9 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 6 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 6: 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

Table 7 lists every rule the released mutator applies, 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.

Table 7: The mutation-rule inventory loaded by the released mutator, by family. [TODO: the released rule files load 120 unique rules (7 definitions duplicate an existing pattern and are skipped); the main text counts 124. Reconcile.]

Rule	Description
Arithmetic (4 rules)	
minus2plus	AOR: - → +
mul2div	AOR: * → /
negate-store	UOI: negate stored expression
plus2minus	AOR: + → -
Indexing (14 rules)	
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
Semantic (21 rules)	
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)
const0to1	init constant 0 → 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 → min
fmin2fmax	min → 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 → 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
Boundary (38 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 → floor-div: tail block never processed
ceil2floor-lit	(x + K-1)/K with literal K → 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
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
lt2le	guard/loop bound $< \rightarrow \leq$: 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
Synchronization (13 rules)	
batchnorm-local-stats-warp-barrier	Mechanism 4 (benign race): uniform random lane statistics are close; per-lane distributions with very different means kill it.

Rule	Description
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
welford-count-merge-early	publish a merged count to the partner lane instead of the consumer
Precision (30 rules)	
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.

Rule	Description
`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: RAW COUNTS AND PROMPTS

Table 8 gives the counts behind the rates of Table 4. All rungs target the same 1,154 official-input survivors across 30 problems. 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.

Table 8: Knowledge ladder, raw counts: survivors killed over survivors evaluated at each rung.

Rung	Killed	Evaluated	Rate
R0 random fuzz	249	1,154	21.6%
R1 naive prompt	495	1,154	42.9%
R2 + taxonomy & ceiling	704	1,154	61.0%
R3 + white-box diffs	626	1,092	57.3%

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.
```