Tools Built in LeanLlama: Purpose, Motivation, and Generalization

This report catalogs the tools built across `leanllama`, explains why each was needed to bridge Lean 4 formalizations with real compiled SPIR-V shaders and Vulkan hardware, and outlines how to generalize them into a reusable verification toolkit.

1. Lean Executables & Reference Interpreters

Tool	Source	Purpose
`shader-model`		Executes the Lean kernel models (`Kernel.*`) on concrete hex word buffers and push constants for differential testing.
`spirv-run`		Interprets real `.spv` binaries directly under `LeanLlama` 's formal SPIR-V semantics (`SPIRV.runStage`), detecting data races, undefined behavior, and unsupported opcodes.
`rmsnorm-model`		Standalone executable runner for the `rms_norm` kernel model.
`cost-counts`		Emits exact hardware event counts (`workgroups` , `dram` , `cache` , `scratch` , `shared` , `barriers` , `subgroup` , `alu` , `threads` , `thread_barriers`) directly from `Cost/RmsNorm*.lean` .

Why We Needed Them

- **Model vs. GPU grounding (`shader-model`):** A Lean specification is only useful if it actually matches what the shader computes. Exporting the Lean models as a CLI binary lets Python differential tests compare GPU outputs bit-for-bit (or within ULP bounds) against the exact Lean definitions.
- **Validating the formal SPIR-V semantics (`spirv-run`):** Before trusting binary-level proofs, we needed to run `SPIRV.Semantics` against hundreds of real `.spv` binaries to find missing opcodes, driver-dependent behavior (`contract` FMA, `sgseq` subgroup ordering, `lenient`

race mode), and actual shader races (e.g., `argsort` F4).

- **Single source of truth for cost fitting (`cost-counts`):** Re-implementing event-counting formulas in Python risks drift between the fitted cost model and the Lean theorems.

How to Generalize

- **Schema-driven `shader-model` dispatch:** Replace manual per-kernel CLI parsing in `ShaderModel/Main.lean` with a uniform JSON/binary descriptor (buffer shapes, dtypes, push-constant structs) derived from `Host/Dispatch.lean` and `Kernels/*`, so adding a new kernel model automatically exposes it to differential testing.
- **Unified `spirv-run` configuration:** Support `f16`/`bf16`/8-bit storage flags and multi-stage pipelines directly from a single manifest rather than per-invocation CLI flags, and share the same `SPIRV` interpreter config with the proof layer (`Step.lean`).
- **Generic `cost-counts`:** Parameterize `CostCounts.lean` over `Cost/HotKernels.lean` (and any `KernelCost` instance) instead of hardcoding the 5 `rms_norm` variants.

2. Vulkan Execution & Differential Testing Harness

Tool	Source	Purpose
`run_shader`	,	Minimal C++ Vulkan compute runner executing any `.spv` with push constants, specialization constants, and storage buffers on GPU or CPU (`RMS_DEVICE=llvmpipe`).
`shaderlib.py`		Shared Python harness driving `run_shader`, `shader-model`, and `spirv-run` (`SPIRV_AS_GPU=1`, `SPIRV_SHADOW=1`).
Differential test suite	`harness/tests/test_*.py`, `run_all.sh`	62 per-family test scripts checking GPU/CPU execution against `shader-model`.
Bug reproducers	, ,	Standalone reproducers for upstream `llama.cpp` / `glslang` findings (`FINDINGS.md`), runnable through `ggml` or directly on CPU Vulkan (`llvmpipe`).

Why We Needed Them

- **Isolating shaders from `ggml`:** Running full LLM graphs obscures single-kernel bugs, subgroup-size assumptions, and rounding differences. `run_shader` lets us test any `.spv` binary in isolation with arbitrary inputs, misalignment offsets, and subgroup requirements.
- **CPU fallback (`llvmpipe`) & CI:** Not every developer or CI runner has an NVIDIA/Intel/AMD GPU. Supporting `RMS_DEVICE=llvmpipe` and `SPIRV_AS_GPU=1` enables deterministic CPU testing in GitHub Actions.
- **Swappable `.spv` directories (`run_test_with_spv.py`):** Allowed us to validate `repro/llama-cpp-fixes.patch` without overwriting the baseline `harness/spv` binaries.

How to Generalize

- **Package `run_shader` + `shaderlib.py` as a standalone library:** Remove hardcoded `llama.cpp` push-constant helpers (`binary_pc`, `unary_pc`) from `shaderlib.py` into a `ggml_abi.py` plugin, leaving `run_shader` and `shaderlib.py` as a generic Vulkan compute differential-testing framework.
- **Consistent device-capability guards:** Enforce a uniform subgroup/feature capability check across all `test_*.py` scripts (fixing skipped-test edge cases like `test_fwht.py` on 8-wide `llvmpipe` subgroups).

3. SPIR-V Coverage, Shadowing, and Gap Analysis

Tool	Source	Purpose
`spirv_shadow.py`	,	Intercepts `run_gpu` calls during tests, re-runs the exact `.spv` under `spirv-run`, and logs `exact` / `diff` / `timeout` / `error` outcomes to JSONL.
`spirv_coverage.py`		Summarizes `SPIRV_SHADOW` logs across all built `.spv` binaries, ranking failure reasons.
`spirv_gap.py`		Statically scans all `.spv` binaries in `harness/spv` against the opcode/GLSL.std.450/type sets supported by `SPIRV/Semantics.lean`.
`coverage.py`		Audits every `.comp` shader in `llama.cpp` across 5 axes (`model`, `spec`, `gpu`, `fp`, `cost`) and generates `COVERAGE.md`.

Why We Needed Them

- **Scaling to 1,500+ compiled binaries:** `llama.cpp`'s `vulkan-shaders-gen` emits >1,500 `.spv` variants. Manual tracking is impossible; `spirv_gap.py` immediately tells us which opcodes block the most shaders statically, while `spirv_shadow.py` + `spirv_coverage.py` show which shaders actually pass or race dynamically.
- **Honest progress accounting (`coverage.py`):** Keeps `COVERAGE.md` synchronized with the actual Lean files and differential tests rather than drifting out of date.

How to Generalize

- **Auto-extract `spirv_gap.py` opcode tables from Lean:** Instead of duplicating `MODULE`, `STEP`, `EVAL`, `SUBGROUP`, and `EXT` sets in Python, emit them from `LeanLlama/SPIRV/Semantics.lean` (or a small Lean reflection command) so the gap checker can never fall out of sync with Lean.
- **Config-driven `coverage.py`:** Move `FAMILIES` and file-mapping rules into a declarative manifest (`coverage.toml`) so any shader repository can generate a multi-axis verification matrix.

4. Code & Proof Generators

Tool	Source	Purpose
`spv_embed.py`		Decodes `.spv` binaries into Lean modules (`Binaries/*.lean`, `*Mod.lean`, `*Lemmas.lean`) with lookup/instruction lemmas.
`binary_proof.py`	,	Generates Lean binary-correctness proofs for all 32 `(op, tA, tB, tD)` variants in the `Binary` family (`add`, `sub`, `mul`, `div` × `{f16, f32}`³).
`rms_copies.py`		Generates the unrolled copy-loop proofs (`Copy2`–`Copy32`) for `rms_norm_f32.spv` (covering `do_multiply` and multi-row workgroup loops).
`iq_tables.py`		Generates Lean lookup tables for IQ quantization formats.

Why We Needed Them

- **Eliminating manual SPIR-V transcription:** Hand-writing SSA IDs, basic block layouts, and hundreds of instruction-lookup lemmas for `.spv` binaries is error-prone and unscalable. `spv_embed.py` turns any `.spv` into a verified Lean AST + lemma set.
- **Scaling proofs across shader families:** When a single `.comp` template compiles into 32 `Binary` `.spv` files or 10 unrolled loop copies inside `rms_norm_f32.spv`, parameterized proof generators (`binary_proof.py`, `rms_copies.py`) let us write one family proof (`Binary/Family.lean`, `RmsNorm/Thread.lean`) and instantiate it mechanically.

How to Generalize

- **Family-parameterized proof generator:** Generalize `binary_proof.py` into a `family_proof.py` engine that takes a family spec (push-constant schema, buffer bindings, scalar/vector conversion hooks, and Lean `Family` theorem) and generates proofs for `Unary`, `GLU`, `Copy`, and `Norm` shader families directly from `spv_embed.py` metadata.
- **AST-driven unrolling in Lean:** Rather than Python-generating `Copy2..Copy32.lean` via string templates, replace unrolled-copy generation with a generic Lean tactic/reflection pass over `Switch` / `Branch` CFG templates where possible, keeping Python generation purely for data tables.

5. Performance Benchmarking & Cost-Model Fitting

Tool	Source	Purpose
`bench.py`, `bench_fusion.py`, `sweep.py`	, ,	Measure GPU dispatch timings across tensor shapes and fused vs. unfused shader pipelines.
`fit_cost.py`, `predict.py`, `compare.py`, `compare_fusion.py`	,	Fit `Dispatch.evalRoofline` weights to measured timings via nonnegative least squares (with leave-one-out cross-validation) and predict fusion speedups.
`microbench/`	`harness/microbench/`	Isolates hardware launch, per-thread, and barrier overheads (`barrier_sweep.py`).

Why We Needed Them

- **Empirical grounding for `Cost/`:** Abstract cost models can't predict fusion profitability or register-cache spilling without calibrating memory vs. barrier/thread weights against real hardware counters/timings.

How to Generalize

- **Target any `Cost/HotKernels.lean` pipeline:** Generalize `sweep.py` + `fit_cost.py` to accept arbitrary dispatch graphs and `cost-counts` schemas so fusion decisions (e.g., `mul_mat` + bias/activation, `rms_norm` + `mul`) can be calibrated on any Vulkan GPU automatically.

6. Proof Purity & CI Automation

Tool	Source	Purpose
`check_proof_purity.sh`		Enforces zero `sorry` / `admit` across Lean sources and verifies via `#print axioms` (`axiom_probe.lean`) that every theorem uses only `propext` , `Classical.choice` , `Quot.sound` , and `Lean.ofReduceBool` .
GitHub Actions CI		Runs `lake build` , `check_proof_purity.sh` , builds `llama.cpp` shaders with LunarG SDK `glslc` , and runs `harness/tests/run_all.sh` on `llvmpipe` .

Why We Needed Them

- **Trustworthy formalization:** Ensures no `sorry` , custom axioms, or silently broken proofs slip in, and continuously tests compiled `llama.cpp` shaders on CPU (`llvmpipe`) even without GPU runners.

How to Generalize

- **AST-based `sorry` check:** Replace `grep -rn -w -e sorry -e admit` (which can false-positive on comments/docstrings) with a Lean environment check inside `axiom_probe.lean` (`Lean.ofReduceBool` / `sorryAx` check) so comments never fail CI while `sorryAx` is caught reliably.
- **Artifact reuse in CI:** Have `purity` and `harness-lavapipe` download the `.lake/build` artifact from `build` instead of rebuilding from scratch.