LUMEN QuantFusion XPU: An Evidence-Aware Architecture, Prototype Evaluation, and Integration Roadmap

Aaron Singh

***Abstract*—This paper presents LUMEN QuantFusion XPU, a project in a twenty-project AI infrastructure portfolio. The current repository is an active prototype with implemented behavior, automated correctness tests, and explicit boundaries around unverified hardware or production claims. We describe the system model, current implementation, goals, and evaluation method, then propose five integrations that advance the project toward reproducible system-level validation. A local audit executed 5 project tests successfully. No accelerator, deployment, or performance conclusion is inferred unless a corresponding committed benchmark or profiler artifact exists.**

***Index Terms*—LUMEN, AI infrastructure, reproducibility, prototype validation, systems evaluation, hardware–software co-design**

I. Introduction

LUMEN QuantFusion XPU (P03) addresses a bounded problem within the portfolio's track-01-gpu-kernels track. Its declared status is *Active Prototype* [1]. The project validates its central mechanism before adding device-specific acceleration, production orchestration, or real-world hardware. This ordering matters because optimization without a trusted reference can make incorrect behavior appear successful.

The project goals are to establish deterministic behavior, encode correctness as tests, create machine-readable evidence, identify limiting resources through measurement, and integrate results with adjacent portfolio systems without losing provenance. The repository contains one paper named exactly after its singular folder: `P03-lumen-quantfusion-xpu.tex`.

II. Technical Context

GPU kernel development requires a correct numerical reference before optimization. Reference equivalence, workload shape, warm-up policy, and profiler evidence are therefore first-class design concerns. The portfolio benchmark standard requires environment, workload, method, metric, artifact, reproduction, and limitation fields; unknown values remain explicitly unmeasured [3].

III. System Model and Architecture

The prototype is organized around the following domain model:

$$s = \frac{\max_i |x_i|}{127}, \quad q_i = \mathrm{clip}(\mathrm{round}(x_i/s), -127, 127), \quad \hat{x}_i = sq_i \tag{1}$$

The equation is a design and test abstraction rather than a claimed empirical law. It supports invariants and expected-value checks while later implementations replace synthetic inputs with representative workloads or devices.

The software architecture contains an input/configuration layer, a deterministic core, validation and evidence output, and a local visualization. The principal inspected source artifacts are `python/quantization_reference.py`. Unsupported real-world conditions are surfaced as limitations rather than silently simulated.

IV. Detailed Script Operation and Rationale

The script generates repeatable floating-point weights, derives a symmetric int8 scale, quantizes and dequantizes the values, and reports MAE, maximum error, RMSE, and estimated raw storage. It isolates quantization error before model accuracy or XPU performance is introduced.

The execution path is:

1) Generate deterministic floating-point input weights.
2) Compute a symmetric scale from the maximum absolute value.
3) Round and clip values into the signed int8 range.
4) Dequantize to approximate floating point.
5) Report reconstruction error and transparent storage estimates.

V. Implemented Prototype

The metadata and source audit found these completed features [2]:

- Python symmetric int8 quantization and dequantization
- Deterministic generated weights
- MAE, maximum error, and RMSE calculations
- Estimated raw storage comparison and unit tests

`python3 -m unittest discover -s tests -p 'test_*.py'` completed successfully with 5 tests. The inspected test artifacts are `tests/unit/test_quantization.py`. This is evidence of local correctness for encoded cases, not production scale or hardware performance.

VI. Testing Methodology and Observed Results

Testing uses Python's standard `unittest` discovery and exercises the public behavior of the reference implementation. The audit reran the suite from the project folder with `python3 -m unittest discover`

TABLE I
IMPLEMENTATION ARTIFACTS AND WHY THEY EXIST

Artifact	Observed responsibility	Engineering rationale
python/quantization_reference.py	quantize_int8, dequantize_int8, error_metrics, make_values, run_experiment, main	Implements the inspectable, unit-tested project core.
scripts/reproduce.sh	Fixed test and demonstration entry point	Gives another developer one command for local reproduction.
PROJECT.yaml and ANALYSIS.md	Status, completed work, planned work, and claim boundaries	Separates declared intent from evidence-backed implementation.
streamlit_app.py	Local evidence and status visualization	Makes outputs inspectable without upgrading simulation into a hardware claim.

`-s tests -p 'test_*.py'`. All 5 discovered tests passed. The result establishes correctness only for the encoded local cases; it does not establish accelerator correctness, real-device behavior, production reliability, or benchmark completion.

No numerical performance result is promoted by this test run. Where scripts emit JSON or JSONL, those outputs remain raw or simulation-specific until a reviewed summary includes hardware, software, workload, warm-up, repetition, correctness threshold, Git revision, and limitations.

VII. Claim Boundaries and Risks

The project records these unverified or excluded claims:

- No model accuracy, runtime speedup, or XPU result has been demonstrated

The main risk is confusing synthetic or modeled behavior with deployed-system behavior. Other risks include incomplete workloads, platform-dependent timing, missing failure injection, and interfaces not yet exercised across device boundaries. Performance claims require a reviewed record meeting the portfolio standard.

VIII. Evaluation Plan

Evaluation proceeds through correctness tests, deterministic reproduction with Git and environment metadata, repeated benchmarks reporting latency/throughput/memory/error metrics, and a named profiler capture tied to exact hardware and source revision. Success requires reference equivalence within a documented tolerance, preservation of safety and resource invariants, clear failures, and evidence reproducible from a clean environment.

IX. Goals, Milestones, and Success Criteria

The project goals are staged so that correctness precedes performance and integration. A goal is complete only when its proof artifact is committed or otherwise reviewable; prose or a simulated number alone is insufficient.

X. Future Work and Integrations

The five project-specific next steps are:

1) Per-channel quantization
2) Small neural-network accuracy comparison
3) PyTorch XPU implementation and profiling
4) Add calibration-aware mixed precision and accuracy–memory tradeoff curves.
5) Connect quantized operators to P02 kernels and P07 serving workloads.

The early items complete declared evidence; the later items connect downstream portfolio consumers. Each integration should add tests and a reviewable artifact such as JSONL evidence, a report, profiler capture, deployment manifest, trace, or labeled data set.

XI. Conclusion

LUMEN QuantFusion XPU is an evidence-aware active prototype: its implemented behavior and tests are real, while unbuilt hardware, deployment, and performance goals remain labeled. Completing the five integrations in dependency order will advance it from a learning artifact toward a credible portfolio component.

References

[1] Aaron Singh, "LUMEN QuantFusion XPU PROJECT.yaml," local portfolio repository.

[2] Aaron Singh, "LUMEN QuantFusion XPU: README, ANALYSIS, source, and test artifacts," local portfolio repository.

[3] Aaron Singh, "AI Infrastructure Portfolio Benchmark Standard," local portfolio repository.

TABLE II
AUDITED TEST MATRIX

Test artifact and case	Behavior being checked	Observed result
tests/unit/test_quantization.py:test_zero_vector_round_trip	Zero vector round trip.	Pass (local)
tests/unit/test_quantization.py:test_values_stay_inside_symmetric_int8_range	Values stay inside symmetric int8 range.	Pass (local)
tests/unit/test_quantization.py:test_reconstruction_error_is_bounded_by_half_scale	Reconstruction error is bounded by half scale.	Pass (local)
tests/unit/test_quantization.py:test_experiment_is_repeatable	Experiment is repeatable.	Pass (local)
tests/unit/test_quantization.py:test_empty_input_is_rejected	Empty input is rejected.	Pass (local)

TABLE III
DECLARED STATUS VERSUS AUDITED EVIDENCE

Audit field	Finding	Evidence source
Declared status	Active Prototype	PROJECT.yaml
Evidence-backed status	Active local prototype; 5 tests passed	Source plus local unittest run
Accepted measured results	None recorded in measured_results	PROJECT.yaml
Mismatch / claim boundary	No model accuracy, runtime speedup, or XPU result has been demonstrated	PROJECT.yaml and ANALYSIS.md
Next proof required	Per-channel quantization; Small neural-network accuracy comparison	Planned features

TABLE IV
PROJECT GOALS AND REQUIRED PROOF

ID	Goal	Completion evidence
G1	Per-channel quantization	Passing tests and a reviewed source artifact
G2	Small neural-network accuracy comparison	Machine-readable result with reproduction metadata
G3	PyTorch XPU implementation and profiling	Reference-equivalence or domain-correctness report
G4	Quantify accuracy–memory tradeoffs	Named profiler, deployment, or integration artifact
G5	Measure XPU runtime using model-shaped tensors	Dashboard/report link preserving provenance and limitations