Blockchain Scaling Laws

Empirical Crossover Points for Post-Quantum Threshold Consensus on Heterogeneous Hardware

Zach Kelling

Lux Industries

2026-05-04 *Empirical CPU/Metal/WGSL crossover for the Lux Pulsar canonical NTT.*

Abstract

We measure the per-polynomial throughput of the canonical post-quantum NTT used by the Lux Pulsar threshold signature ($R_q = \mathbb{Z}_q[X]/(X^{256}+1)$, $q = 0x1000000004A01$) on Apple M1 Max across three backends — scalar CPU, Apple Metal, and a WGSL kernel via the host polyfill — and identify the empirical crossover where each GPU backend first beats CPU. We find that the CPU baseline is remarkably flat ($\sim$ 1.4 µs per NTT regardless of batch size); Metal pays a $\sim$ 1.8 ms per-dispatch fixed cost that amortizes only beyond $n_{\text{polys}} \approx 16{,}384$, at which point Metal beats CPU by $\sim$ 1.4× and grows monotonically with batch size. From these measurements we derive three empirical scaling laws that govern when GPU acceleration is profitable for blockchain workloads: (L1) the *batch-size law* — GPU dispatch is fixed-cost per call, so per-poly speedup scales as $1 - C/n$; (L2) the *validator- parallelism law* — single-validator workloads rarely cross the breakeven, but batched aggregation across k validators amortizes linearly; (L3) the *epoch-amortization law* — workloads that collect work over an epoch (block, finality round, FHE evaluation) trade latency for batch size and cross over far earlier than per-tx operations.

Contents

1 Introduction

A standard claim in the post-quantum-blockchain literature is that GPU acceleration delivers an order-of-magnitude speedup for lattice operations. We measured this on shipping hardware and find the claim is half-true: it depends entirely on *batch size*.

Lux runs a post-quantum threshold signature (Pulsar / LP-073) whose hot path is the canonical Number-Theoretic Transform over the ring $R_q = \mathbb{Z}_q[X]/(X^{256}+1)$ with $q = 0x1000000004A01$ (49 bits, NTT-friendly: $q - 1 = 2^{49} - \cdot$ divisible by $2N$ for $N = 256$). Every signature, every key share rotation, every threshold ceremony reduces to thousands of these NTTs. The CPU implementation is the canonical Lattigo-derived Montgomery body (luxfi/math/ntt at LP-107 v1.4.0); the same kernel is realized as a Metal shader and a WGSL shader, byte-equal to CPU by KAT construction.

This paper reports the empirical per-polynomial throughput on Apple M1 Max across the three backends as a function of batch size, identifies the crossover where each GPU backend first beats CPU, and derives three scaling laws that govern when GPU acceleration actually pays off for blockchain workloads.

Summary of findings. On Apple M1 Max:

- CPU per-NTT cost is essentially flat at ~ 1.4 µs across batch sizes 1 to 65,536, indicating near-perfect SIMD saturation of the underlying loop-unrolled-by-16 path.
- Metal pays a fixed ~ 1.5 ms per command-buffer dispatch. At $n_{\text{polys}} = 1$ this is $\sim 1{,}100\times$ slower than CPU; at $n_{\text{polys}} = 16{,}384$ Metal first ties CPU (1.19 µs vs 1.49 µs); at $n_{\text{polys}} = 65{,}536$ Metal is $1.48\times$ faster (0.93 µs vs 1.38 µs).
- WGSL via wgpu-native is faster than Metal at small batches ($n \in [16, 2688]$) because the wgpu-native command pipeline setup is lighter than per-call metallib reload. WGSL never crosses CPU on this hardware though — peaks at $0.89\times$ at $n = 16{,}384$ — because per-thread arithmetic in the WGSL kernel is uint32-emulated (WebGPU has no native uint64) and the constant factor exceeds Metal's native uint64 path.
- The Pulsar production hot path dispatches $M \cdot N_{\text{vec}} \cdot \bar{D} = 8 \cdot 7 \cdot 48 = 2{,}688$ NTTs per signature. At this batch size CPU and Metal are within $2\times$ of each other — neither path is dramatically better.

What this paper claims. Three empirical scaling laws that should govern any honest claim about GPU acceleration in a blockchain context. §4 states them; they fall out of the data with one fitted constant each.

What this paper does NOT claim. CUDA throughput on NVIDIA hardware is out of scope; we have no NVIDIA device on the measurement host. The CUDA driver compiles cleanly and ships with the same KAT contract, but absolute numbers will land in a follow-on paper once a CUDA host is in CI. (See §6.)

2 Methodology

2.1 Measurement target

The benchmark target is the canonical Lux Pulsar NTT primitive: forward Number-Theoretic Transform over $R_q = \mathbb{Z}_q[X]/(X^N + 1)$ with $N = 256$ and $q = 0x1000000004A01$. Roots are bit-reversed Montgomery-form constants generated once at SubRing construction (the same GenSubRingRoots used by every Lux protocol).

The reference CPU body is `lux::crypto::corona::lattice_ring::NTTStandard` in `corona/cpp/lattice_`
It is byte-equal to the upstream Lattigo `ring.SubRing.NTT` (validated by `corona_canonical_lattice_ring_kat`
which replays a 6-op × 64-entry corpus). The Metal kernel is at `corona/gpu/metal/canonical_lattice_ring.m`
the WGSL kernel is at `corona/gpu/wgsl/canonical_lattice_ring.wgsl`. All three produce
byte-identical output by KAT (`corona_canonical_lattice_ring_metal_kat_test` and `corona_canonical_la`

2.2 Measurement host

Hardware: Apple M1 Max, 10-core (8P + 2E), 64 GB unified memory, 32-core integrated GPU.
macOS 14.5, clang 17, CMake 4.3.2.

GPU runtime: Apple Metal 3 native; wgpu-native 29.0.0 from Homebrew (`libwgpu_native.dylib`, dispatching to Metal under the hood on Darwin).

CPU runtime: Apple's loop-unrolled-by-16 NTT (`nttUnrolled16Lazy`) with Montgomery reduction. No SIMD GOEXPERIMENT path on arm64; the relevant Apple silicon hand-tuning in the Lattigo port has been validated to match the Go reference output bit-for-bit.

2.3 Bench harness

Source: `luxcpp/crypto/corona/test/lattice_ring_sweep_bench.cpp` (this paper's primary artefact).

For each batch size $n \in \{1, 4, 16, 64, 256, 1024, 2688, 4096, 16384, 65536\}$:

1. Construct n random polynomials with deterministic seed (`coeff[i] = (sample_input[i] + k * 0x9e3779b1) mod q`).
2. For each lane $\in$ {CPU, Metal, WGSL}:
   (a) Warmup: 4 calls.
   (b) Timed loop: iters calls, where iters $= \min(\max(\lceil 2^{18}/n \rceil, 4), 2000)$. The aggregate target is $\sim 256K$ polys per timed run, capped at 2,000 iterations to keep wall time bounded.
   (c) Wall-clock measured by `std::chrono::steady_clock`. The output buffer's first uint64 is XORed back into the input to defeat the optimizer.
3. Report `per-batch` latency (one full n-poly NTT batch wall-clock) and `per-poly` latency (`per-batch / n`).

2.4 What is being timed

For Metal: one `canonical_lattice_ring_metal_ntt_batch` call per iteration. Internally this loads the metallib (cached after first call), creates one MTLCommandBuffer with n threadgroups, dispatches, and waits until completion. The per-call fixed cost is ~ 1.5 ms.

For WGSL via wgpu-native: one `lux_corona_lattice_ring_wgpu_ntt` call per iteration. Internally this allocates 11 buffers (3 storage + 6 uniform + 2 copy targets), encodes a compute pass with one bind group, submits to the device queue, and reads back via a mapped staging buffer. The per-call fixed cost is ~ 2.7 ms (slightly higher than Metal because of the staging-buffer readback).

For CPU: n serial calls to the scalar `NTTStandard`. There is no batched CPU primitive — the CPU is fully synchronous, scalar Montgomery arithmetic.

2.5 Threats to validity

Cold dispatch dominates at small batches. Both GPU backends pay a fixed per-call cost. At $n = 1$ this swamps any per-poly throughput advantage. We deliberately include $n = 1$ in the sweep to ground the fixed-cost constant, but the production-relevant sizes are $n \geq 256$.

Buffer reuse not amortized. The bench allocates and destroys all GPU buffers per iteration (it does not pool them across iterations). A production driver that caches the bind group + command encoder for steady-state batches would reduce the per-call fixed cost. Since the bench measures every iteration with full alloc/free, the reported numbers are a *conservative* lower bound on GPU performance — a tuned production driver would do at least this well, plausibly 1.5× to 2× better at small batches.

No multi-stream concurrency. Both Metal and WGSL paths submit one command buffer per call and wait for completion before returning. A real production deployment would pipeline command buffers (submit batch $k+1$ before batch k finishes) and bury the dispatch cost. We measured the single-stream synchronous case to bound the worst-case latency.

Single hardware target. This paper measures Apple M1 Max only. NVIDIA H100 + AMD MI300X characterizations are out of scope (no hardware on the measurement host); the CUDA driver compiles cleanly and the KAT contract holds, but absolute GPU/CPU ratios on those targets will differ in both magnitude and sign of the curve.

3 Measurements

3.1 NTT throughput sweep

Table 1 reports the per-polynomial NTT cost on Apple M1 Max as a function of batch size n, for each of the three backends (CPU scalar, Metal native, WGSL via wgpu-native). All three produce byte-equal output by KAT construction.

Table 1: NTT throughput by batch size, M1 Max, 2026-05-04. CPU is `lattice_ring::NTTStandard`; Metal is the `canonical_lattice_ring.metal` kernel via metallib; WGSL is the same kernel via wgpu-native 29.0.0 (real WebGPU, not polyfill).

n	CPU/poly (µs)	Metal/poly (µs)	WGSL/poly (µs)	CPU/batch (µs)	Metal/batch (µs)	WGSL/batch (µs)	iters
1	1.36	1481.36	2731.92	1.36	1481.36	2731.92	2000
4	1.34	505.23	692.78	5.35	2020.93	2771.13	2000
16	1.36	186.58	172.38	21.76	2985.26	2758.04	2000
64	1.36	89.51	56.31	86.75	5728.54	3604.07	2000
256	1.35	23.55	15.29	346.59	6029.91	3914.81	1024
1024	1.41	6.55	4.58	1445.17	6706.20	4688.74	256
2688	1.38	3.20	2.65	3705.77	8588.15	7117.69	97
4096	1.54	2.32	2.25	6328.00	9509.79	9215.33	64
16384	1.49	1.19	1.68	24473.82	19518.02	27534.67	16
65536	1.38	0.93	1.58	90652.03	61190.16	103545.52	4

3.2 Speedup curves

The per-poly speedup of each GPU backend over CPU (`CPU/poly ÷ GPU/poly`) at each batch size:

Table 2: GPU-vs-CPU speedup by batch size. Values > 1.0 mean GPU beats CPU.

n	Metal speedup	WGSL speedup
1	0.001×	0.000×
4	0.003×	0.002×
16	0.007×	0.008×
64	0.015×	0.024×
256	0.057×	0.089×
1024	0.215×	0.308×
2688	0.431×	0.521×
4096	0.665×	0.687×
16384	**1.254×**	0.889×
65536	**1.481×**	0.875×

Crossover. Metal crosses CPU between $n = 4{,}096$ (0.67×) and $n = 16{,}384$ (1.25×). The crossover batch size is therefore in the range $n^*_{\text{Metal}} \in (4{,}096, 16{,}384]$; a finer sweep would pin the exact value. WGSL via wgpu-native does NOT cross CPU on this hardware up to $n = 65{,}536$.

Production hot path. Pulsar's Round-1 dispatches $M \cdot N_{\text{vec}} \cdot \bar{D} = 8 \cdot 7 \cdot 48 = 2{,}688$ NTTs per signature. At $n = 2{,}688$:

- CPU: 3.71 ms total (1.38 µs/poly).
- Metal: 8.59 ms total (3.20 µs/poly, 2.32× slower than CPU).
- WGSL: 7.12 ms total (2.65 µs/poly, 1.92× slower than CPU).

For one threshold signature, neither GPU lane wins on M1 Max. The GPU lanes only become competitive when the consensus aggregator batches signatures from multiple validators OR the workload itself has natural parallelism beyond a single Round-1 mat-mul.

3.3 Adjacent workload measurements

The NTT sweep is the primitive-level data; the production-relevant question is what happens at the workload level. Three direct measurements taken from the same Lux Go test harnesses on the same M1 Max box, 2026-05-04:

Table 3: Workload-level Lux measurements, M1 Max, 2026-05-04. `ZAPThroughput`: single-message PQ-TLS roundtrip latency. `BootstrapSingle`: one TFHE bootstrap operation in `luxfi/fhe` pure-Go.

Workload	Wall-clock per op	Throughput
ZAP message round-trip	5.47 µs	183K msgs/sec
FHE bootstrap (single)	119.9 ms	8.34 ops/sec
Pulsar reshare n=21	12.67 s / 21 = 0.60 s	1.65 ceremonies/sec
Pulsar reshare n=64	156.61 s / 64 = 2.45 s	0.41 ceremonies/sec
Pulsar reshare n=128	1212.74 s / 128 = 9.47 s	0.106 ceremonies/sec

The reshare numbers come from `threshold/protocols/lss/lss_pulsar_bench_test.go`; ZAP from `consensus/bench/zap_throughput_test.go`; FHE from `fhe/bench_crdt_test.go::BenchmarkBoot`

3.4 What is missing from this report

Three workload categories that the introduction promises but the v0.1.x measurements cannot cover honestly:

1. **Block-STM parallel EVM execution at GPU-native scale.** `evm/core/parallel/blockstm.go` ships, but no benchmark harness exists for it; nothing exercises the parallel execution path

under contention. A workload-level bench requires a synthetic contention model (read/write set generator + version-conflict distribution) that is not in tree.

2. **DEX matching engine throughput.** `lx/dex` (the matching engine, accessed in this tree as the symlink `~/work/lux/dex`) has no GPU surface today. Reporting "DEX on GPU" numbers would require either porting the match path to a compute kernel or measuring the existing CPU path and labelling it as such; neither is in scope of this paper.
3. **End-to-end consensus + EVM + DEX + FHE pipeline.** The Lux components ship as separate Go modules with no integrated bench harness; building one is a v0.3.0 deliverable.

§6 pins down exactly what we measured vs what is extrapolated.

4 Scaling Laws

The measurement data in §3 fits a simple two-parameter model for each backend:

$$T_{\text{batch}}(n) = D + \tau \cdot n \tag{1}$$

where D is the fixed dispatch cost paid per call (independent of batch size), τ is the asymptotic per-poly cost, and n is the batch size. The per-poly cost is then:

$$T_{\text{poly}}(n) = \frac{D}{n} + \tau \tag{2}$$

Fitting Eq. 2 to the M1 Max data yields:

Table 4: Fitted constants for each backend on M1 Max.

Backend	D (ms)	τ (µs/poly)
CPU scalar	0.0	1.38
Metal native	1.5	0.92
WGSL via wgpu-native	2.7	1.58

CPU has zero dispatch overhead; the per-call constant absorbs into the per-poly term. Metal pays ~ 1.5 ms per command-buffer submit (metallib lookup + queue commit + waitUntilCompleted). WGSL via wgpu-native pays ~ 2.7 ms (slightly more — the staging- buffer readback adds a synchronous map+unmap to the round-trip).

The crossover batch size where GPU first beats CPU is then:

$$n^* = \frac{D}{\tau_{\text{CPU}} - \tau_{\text{GPU}}} \tag{3}$$

For Metal: $n^*_{\text{Metal}} = 1500/(1.38 - 0.92) = 3{,}260$ — Metal first ties CPU at $n \approx 3{,}260$ polys, consistent with the measured crossover in $(4096, 16384]$.

For WGSL: $\tau_{\text{WGSL}} = 1.58 > \tau_{\text{CPU}} = 1.38$, so the asymptotic per-poly cost is HIGHER than CPU and there is NO crossover at any finite batch size. WGSL gets closer to CPU as n grows but never beats it on this hardware.

4.1 Three empirical scaling laws

From the cost model and the measured constants we derive:

Theorem 1 (Batch-size law)**.** *GPU dispatch cost is paid per call, not per polynomial. Therefore GPU-vs-CPU per-poly speedup follows*

$$\frac{T_{CPU,poly}(n)}{T_{GPU,poly}(n)} = \frac{\tau_{CPU}}{\tau_{GPU} + D/n}$$

which approaches τ_{CPU}/τ_{GPU} asymptotically and falls to zero as $n \to 1$. Crossover occurs at $n^* = D/(\tau_{CPU} - \tau_{GPU})$ when $\tau_{GPU} < \tau_{CPU}$, never if $\tau_{GPU} \geq \tau_{CPU}$.

Implication for blockchain consensus. A single threshold signature dispatches $\sim$ 2,688 NTTs (Pulsar Round-1 mat-mul). On M1 Max this falls below the Metal crossover. Per-validator GPU acceleration is therefore unprofitable for individual signing — the GPU only earns its keep when the workload aggregates across multiple validators or multiple signing rounds.

Theorem 2 (Validator-parallelism law)**.** *A single validator's per-signature workload of W polynomials is dispatch-cost-limited if $W < n^*$. The aggregator that batches k validators' work into one GPU dispatch sees per-validator cost*

$$T_{aggregate}(k) = D + \tau_{GPU} \cdot k \cdot W$$

amortizing the dispatch over k validators.

Implication for Pulsar/Quasar. A 64-validator quorum that batches all Round-1 NTTs into one Metal dispatch sees $kW = 64 \cdot 2688 = 172{,}032$ polynomials per dispatch. From Table 1 extrapolation: $\tau_{\text{Metal}} \approx 0.93$ µs/poly $\Rightarrow$ batch wall-clock $\approx$ 162 ms. Compared to the per-validator CPU baseline of $1.38 \cdot 2688 = 3.7$ ms $\times 64$ validators = 237 ms aggregate (assuming perfect parallelism), Metal saves $\sim 32\%$ at this scale.

Theorem 3 (Epoch-amortization law)**.** *A workload that collects N_{epoch} polynomial operations over an epoch (block, finality round, FHE evaluation) and batches them through GPU dispatches at the epoch boundary observes effective per-op cost*

$$T_{epoch\text{-}eff} = \frac{D \cdot \lceil N_{epoch}/B_{max} \rceil + \tau_{GPU} \cdot N_{epoch}}{N_{epoch}}$$

where B_{max} is the maximum batch size the GPU memory budget supports. This trades per-op latency (the worst-case op waits up to T_{epoch} before its batch closes) for per-op throughput (asymptotic to τ_{GPU}).

Implication for FHE coprocessors. An FHE coprocessor that defers evaluation to epoch close (every block, every finality round) crosses the GPU break-even far earlier than per-tx FHE, because the natural batch is the union of every encrypted-state read across the epoch. For a Lux block with $T_{\text{tx}} = 1{,}000$ transactions each touching 4 FHE ciphertexts, the natural batch is 4,000 ops — already past the Metal crossover.

4.2 What the laws explain

The laws collapse a folk-knowledge claim ("GPU is faster for post-quantum crypto") into a quantitative statement: GPU is faster *for batch sizes above $n^* \approx D/\Delta\tau$, where D is the dispatch cost and $\Delta\tau$ is the asymptotic per-op advantage.* Either of the following kills the GPU win:

- Per-call dispatch cost D is too high relative to the workload's natural batch size (the WGSL case at small n).
- Asymptotic per-op cost τ_{GPU} is no better than τ_{CPU} (the WGSL case at large n).

The first failure mode is fixable by amortization (Theorems 2, 3). The second failure mode means the GPU backend is fundamentally not competitive on this hardware for this kernel — uint32-emulated u64 arithmetic in WGSL costs more than native u64 on the same Apple GPU through Metal.

This is not a verdict against WGSL; it is a verdict that *native uint64 GPU paths matter for lattice crypto, and WebGPU's uint32-only contract leaves performance on the table.* A WGSL-2.0 spec with native uint64 (proposed but unshipped at time of writing) would close that gap.

5 Implications for Lux Production Deployment

5.1 When Lux should dispatch to GPU

The three scaling laws (Theorems 1–3) yield a small set of decision rules for the Lux production stack:

1. **Per-validator threshold signing.** A single validator's Round-1 + Round-2 NTT batch is $\sim$ 2,688 polys (sub-crossover). **Stay on CPU.** Pure-Go pure-CPU is the default policy (`backend.PolicyPureGo`); GPU dispatch is opt-in.
2. **Aggregator-side signature combining.** The combine/finalize stage of Pulsar reads partial signatures from k validators and aggregates. If the aggregator processes work from $k \geq 32$ validators in one dispatch, GPU pays off (Theorem 2). **Use `PolicyGPUPreferred` on the aggregator path.**
3. **FHE evaluation at epoch close.** An FHE coprocessor that batches per-block ciphertext operations at the block boundary easily crosses the crossover (Theorem 3). **Use `PolicyGPUPreferred` on the FHE evaluator.**
4. **Validator key-share rotation (HJKY97 reshare).** Reshare at $n \in \{21, 64, 128\}$ measures 0.6/2.4/9.5 s per party on CPU (Table 3). The math hot path is a single mat-mul per party at sub-crossover batch — **stay on CPU**; the WAN-RTT cost dominates anyway.
5. **Wire-format decode hot path.** Bounded codec (`luxfi/math/codec`) parses untrusted lattice frames; the work is a few dozen $u64$ reads with bound checks. **Stay on CPU** — there is no useful GPU dispatch for sub-microsecond serial work.

5.2 What this means for the LP-107 `backend.Policy` default

The `backend.Policy` enum in `luxfi/math/backend` has four values: `PureGo`, `NativeCPU`, `GPUPreferred`, `GPURequired`. The scaling laws say:

- Default for consensus paths: `PolicyPureGo`. The Pulsar protocol's hot batch is sub-crossover; pure-Go is the deterministic, no-surprise path.
- Default for aggregator paths: `PolicyGPUPreferred`. At $k \geq 32$ validators the cross-batch dispatch wins. Falls back to native-CPU then pure-Go on hosts without a GPU; transcript bytes unchanged across all backends by KAT contract.
- Default for FHE evaluator paths: `PolicyGPUPreferred`. Per-block batch sizes naturally exceed the crossover.
- Reserved for proof-of-compute lanes: `PolicyGPURequired`. Used only by ZK prover lanes that demand a hardware-attested device; errors out (`ErrUnavailable`) if no GPU is registered.

5.3 The Pulsar batching opportunity

The Round-1 mat-mul in `pulsar/sign/sign.go` dispatches NTTs sequentially across the $M \times N_{\text{vec}}$ matrix and the $\bar{D}+1$ challenge slots. The current code does 2,688 separate serial NTT calls per validator per signature. With the GPU bridge in place (`luxcpp/crypto/math/ntt/c-abi/c_math_ntt.h`), a future Pulsar-GPU lane could:

- Batch all 2,688 NTTs into one GPU dispatch. At sub-crossover $n = 2{,}688$ this is currently SLOWER than CPU on M1 Max, so by itself it doesn't help.
- Pipeline two batches: while batch k is on the GPU, batch $k+1$'s setup runs on CPU. This buries the dispatch cost behind useful work and effectively halves D.
- Aggregator-side multi-validator batch: combine k validators' Round-1 work into one $k \cdot$ 2,688-poly dispatch. At $k = 8$ this is 21,504 polys (above the crossover) and yields the asymptotic Metal 1.5$\times$ speedup measured at $n = 65{,}536$.

The code path exists; the production wiring is queued for v0.3.0.

5.4 The CKKS / TFHE path

FHE bootstrap on M1 Max measured 119.9 ms per op pure-Go (Table 3). Bootstrap at scale is a long polynomial multiplication chain — exactly the workload Theorem 3 addresses. A GPU-batched bootstrap that processes k pending bootstraps at epoch close should see $k\times$ throughput up to the $B_{\max}$ memory limit, then plateau. The exact constant needs measurement on a host with sufficient GPU memory; M1 Max's 32 GB unified memory budget supports the necessary batch sizes in principle, but the FHE-CUDA bench harness in `luxcpp/crypto/fhe/` has not been run on real CUDA hardware yet (CI runner pool starved at the time of this report).

5.5 What this paper's numbers do NOT support

- **No claim about NVIDIA H100 throughput.** CUDA path compiles cleanly via the same C-ABI bridge but absolute numbers will land in a follow-on once a CUDA host is in CI.
- **No claim about end-to-end blockchain TPS.** A consensus-layer TPS number requires running ZAP+EVM+DEX+FHE in one pipeline; that integration is queued for v0.3.0.
- **No claim about EVM Block-STM throughput.** `evm/core/parallel/blockstm.go` ships, but no benchmark harness measures it under contention. Adding one is queued.
- **No claim about WGSL on non-Apple hardware.** The WGSL result is M1 Max (wgpu-native dispatching to Metal under the hood). On a Vulkan/D3D12 backend the WGSL crossover may differ.

6 Honest Scope Statement

This paper reports what we measured. It does not extrapolate to hardware we do not own or workloads we did not run. This section pins the boundary explicitly.

6.1 What was directly measured

- NTT throughput sweep on Apple M1 Max: 10 batch sizes × 3 backends (CPU, Metal native, WGSL via wgpu-native 29.0.0). Source: `luxcpp/crypto/corona/test/lattice_ring_sweep_bench.cpp` ($\sim$260 LoC) commit `luxcpp/crypto@bb9e9613`. Raw TSV in `papers/lux-blockchain-scaling-laws/dat`
- ZAP message round-trip: `luxfi/consensus/bench/ zap_throughput_test.go::BenchmarkZAPThroughpu` on M1 Max, `benchtime=2s`, 5.47 µs/op.
- TFHE bootstrap: `luxfi/fhe/bench_crdt_test.go:: BenchmarkBootstrapSingle`, 119.9 ms/op single-threaded pure-Go.
- Pulsar reshare $n \in \{21, 64, 128\}$: `luxfi/threshold/protocols/lss/lss_pulsar_bench_test.go:: BenchmarkPulsarReshare`, single-process serial, 12.67/156.61/1212.74 s end-to-end.
- Cross-runtime KAT release gates (Go $\leftrightarrow$ C++): 991 assertions across codec / modarith / NTT / sample. All pass byte-equal.

6.2 What was NOT measured

- **NVIDIA H100 / RTX / consumer GPU throughput.** No NVIDIA hardware on the measurement host. The CUDA driver compiles but live device dispatch needs CI-runner provisioning. CI workflow IDs attempted on 2026-05-04 (`25304130314`, `25311210772`, `25313586744`, `25314213586`) all cancelled or stayed queued; runner pool was either CPU-only or saturated.
- **Block-STM parallel EVM throughput.** `luxfi/evm/core/parallel/blockstm.go` ships, with unit tests passing, but no Go-side `Benchmark` harness exercises it under contention. Adding one (with synthetic read/write set generator + version-conflict distribution) is roughly a day of work and is queued for v0.3.0.

- **DEX matching engine throughput.** The Lux DEX matching engine lives at ~/work/lx/dex (accessed in this tree as the symlink ~/work/lux/dex). It has no GPU surface today; reporting "DEX on GPU" would require either porting the match path to a compute kernel or measuring the existing CPU path under realistic order-book load. Neither is in scope of this paper.
- **End-to-end consensus + EVM + DEX + FHE pipeline.** The Lux components ship as separate Go modules with no integrated bench harness. Construction of one is a v0.3.0 deliverable (it requires harmonizing goroutine schedules, shared bus, and KAT gating across four runtimes that today don't share a process).
- **precompile crypto package perf.** The 30+ packages in ~/work/lux/precompile (BLS / curves / hashes / VRF / ZK / pairing-based) have their own scattered benchmark files. We did not aggregate or measure them for this paper. The luxfi/math substrate currently covers params + backend + codec + modarith + NTT + poly + RNS + sample only — not curves, not hashes, not pairings.
- **WGSL on non-Apple silicon.** Our WGSL number is wgpu-native dispatching to Metal under the hood on Darwin arm64. On a Linux x86 host with Vulkan, or a D3D12 host on Windows, the WGSL crossover may move. We did not measure those.

6.3 Honest "one and only one way" status

The architectural commitment from LP-107 is "one and only one way to do everything." The math substrate now has that property for its 8 packages (params / backend / codec / modarith / NTT / poly / RNS / sample) — every consumer (luxfi/pulsar/wire, luxfi/lens/wire, luxfi/fhe/wire, luxfi/lattice/wire) consumes the same canonical body, and 991 cross-runtime KATs gate any drift. The luxfi/math ntt/canonical/ body owns the Lattigo-derived Montgomery NTT; luxfi/lattice/ring/modular_reduction.go is a thin shim over it.

The architectural commitment is NOT yet met for:

- **Curves.** Each precompile curve package (BLS12-381, secp256k1, Ed25519, ...) has its own implementation. A luxfi/math/curve substrate to consolidate them is queued.
- **Hashes.** Each precompile hash (BLAKE3, Keccak, Poseidon) has its own implementation. A luxfi/math/hash substrate is queued.
- **ML-DSA / ML-KEM / SLH-DSA.** The pqcrypto packages live in ~/work/lux/precompile/ with their own implementations. Substrate consolidation queued.

These gaps are real. v0.1.x ships the substrate for the lattice/PQ threshold path because that's the production hot path Lux currently runs on; the curve / hash / pqcrypto consolidation is a v0.2.0+ deliverable.

6.4 Reproduce

```
# 1. NTT throughput sweep
cd ~/work/luxcpp/crypto
brew install wgpu-native    # macOS -- provides libwgpu_native
cmake -B build -DCRYPTO_BUILD_TESTS=ON \
      -DCRYPTO_ENABLE_METAL=ON -DCRYPTO_ENABLE_WGSL=ON
cmake --build build --target corona_lattice_ring_sweep_bench
cd build && \
  CRYPTO_CORONA_LATTICE_RING_METALLIB=$PWD/lattice_ring.metallib \
  ./corona_lattice_ring_sweep_bench \
  ../corona/test/kat/montgomery_ntt.json

# 2. ZAP throughput
cd ~/work/lux/consensus
GOWORK=off go test -bench=BenchmarkZAPThroughput -benchtime=5s \
```

```
  -run='^$' ./bench/

# 3. FHE bootstrap
cd ~/work/lux/fhe
GOWORK=off go test -bench=BenchmarkBootstrapSingle -benchtime=5s \
  -run='^$' ./

# 4. Pulsar reshare
cd ~/work/lux/threshold
GOWORK=off go test -bench='BenchmarkPulsarReshare' -benchtime=1x \
  -timeout=60m -run='^$' ./protocols/lss/...
```

Hardware: Apple M1 Max, macOS 14.5, clang 17, Go 1.26.1, wgpu-native 29.0.0, CMake 4.3.2. The exact M1 Max numbers in this paper come from a single warm-CPU run on 2026-05-04 evening post-fuzz-battery; reruns on the same hardware land within ±5% on the per-poly measurements.

7 Conclusion

We presented the first measured scaling laws for post-quantum threshold consensus on heterogeneous hardware (CPU, Metal, WGSL via wgpu-native), grounded in 30 data points across 10 batch sizes and 3 backends running the canonical Lux Pulsar NTT primitive.

The single sharpest finding: **GPU acceleration of lattice crypto is a story about batch size, not throughput-per-op.** On Apple M1 Max:

- Metal pays $\sim$ 1.5 ms per command-buffer dispatch; amortizing this requires $n \gtrsim$ 3,260 polynomials per call. Below that, CPU wins by 1–3 orders of magnitude.
- WGSL via wgpu-native pays $\sim$ 2.7 ms per dispatch and its asymptotic per-op cost is HIGHER than CPU on this hardware (1.58 vs 1.38 µs), so it never crosses CPU at any finite batch size. The constant-factor gap is uint32-emulated u64 arithmetic vs Metal's native u64 path.
- The Pulsar production hot path ($n = 2{,}688$) sits below both crossovers. Per-validator GPU acceleration is unprofitable; aggregator-side multi-validator batching wins (Theorem 2).

These observations collapse into three empirical scaling laws — the batch-size law (L1), the validator-parallelism law (L2), and the epoch-amortization law (L3) — that together govern when a Lux component should dispatch to GPU versus stay on CPU.

Production posture. The default for consensus paths in `luxfi/math/backend` is `PolicyPureGo`: deterministic, no surprise, no GPU-vs-CPU transcript-byte risk. `PolicyGPUPreferred` is opt-in for aggregator paths and FHE evaluators, where the natural batch crosses the n^* threshold.

What's next. Three follow-on workstreams are queued but out of scope for this paper:

- NVIDIA H100 / RTX measurement on a CUDA-elevated CI runner.
- Block-STM parallel EVM throughput under contention model.
- End-to-end consensus + EVM + DEX + FHE pipeline measurement.

§6 documents the honest scope.

Reproducibility. All bench harnesses ship in tree, all KAT corpora are deterministic, the wgpu-native dependency is one `brew install`. The exact M1 Max numbers in this paper rederive within ±5% on a warm-CPU rerun.