DAG-EVM and Conflict-Graph Consensus for the Lux Multi-Chain

Zach Kelling

Lux Industries

March 2026

Abstract

We present a unified DAG consensus model for the Lux multi-chain that lifts Block-STM-style optimistic concurrency from a per-block scheduler into a first-class finality primitive. Each block-producing chain (C-Chain EVM, D-Chain DEX, Z-Chain ZK-UTXO, A-Chain AI, O-Chain Oracle, R-Chain Relay, S-Chain ServiceNode) emits transaction batches as vertices in a native directed acyclic graph, with conflict sets derived from the chain's natural state model: EVM read/write sets, UTXO nullifiers, DEX order tuples, job identifiers, feed rounds, destination-chain nonces, or service epochs. Antichains are finalised in parallel under the Quasar post-quantum certificate scheme; sequential worldview is preserved by a serialisability theorem. We formalise the safety and no-double-spend properties in Lean 4 and TLA$^+$, prove commutativity of non-conflicting vertices, and show that GPU-native conflict-graph reduction plus GPU ecrecover/keccak yields a 32× practical speedup on the dominant critical-path operations. The result is a DAG-EVM that preserves bytecode-level compatibility with Ethereum, a DAG-Z-Chain that preserves ZK-UTXO double-spend security, and a framework that cleanly contrasts Lux against contemporary parallel and DAG blockchains (Aptos, Sui, Solana, Monad, Fuel).

Contents

1 Introduction

Parallel blockchain execution splits into two orthogonal problem families: *intra-block* parallelism (execute the transactions of a linear block in parallel) and *inter-block* parallelism (finalise non-conflicting blocks in parallel, a partial-order consensus). The industry has focused almost exclusively on the first: Aptos' Block-STM [1], Monad's pipelined execution [3], Sei's parallel EVM, and Polygon Miden's optimistic execution all run in linear-chain consensus. DAG consensus, when present (Sui's Mysticeti [2], Aleph Zero's AlephBFT, Avalanche's Snowman^{++}), typically does not thread the DAG into the state-machine layer: blocks are ordered by the DAG and then executed sequentially, so the DAG exists above the execution engine rather than inside it.

Lux takes the third path: *the execution engine's conflict graph **is** the consensus DAG.* Each VM emits vertices whose parents are determined by the state-level dependency relation of that VM, and the DAG engine (nebula) finalises antichains that by construction contain no mutually-conflicting vertices. The downstream property is **serialisability by construction**: any topological order of the accepted DAG yields identical state, and no two finalised siblings can double-spend, re-write the same EVM slot, or re-submit the same nullifier.

This paper presents:

- A uniform definition of the per-chain conflict rule (§2);
- The DAG-EVM construction and its Block-STM lineage (§3);
- The DAG-Z-Chain nullifier-conflict construction (§4);
- Lean 4 formal proofs of safety, commutativity, and serialisability (§5);
- TLA$^+$ specifications and model-checked invariants (§6);
- A principled comparison with contemporary parallel and DAG blockchains (§7);
- A discussion of GPU-native EVM execution covering the v0.20.0 $\rightarrow$ v0.23.0 release cycle, including spec-complete Metal and CUDA opcode kernels, the EVMC host bridge for `CALL`/`CREATE` pre-resolution, and a unified pipeline that runs EVM execution, consensus state hashing, and post-quantum signature verification on a single Metal device (§8);
- A four-way parity test that runs the same 133-vector bytecode corpus through CPU_Sequential, CPU_Parallel (Block-STM), GPU_Metal and GPU_CUDA, asserting bit-exact agreement on `(gas_used, status, output)` (§10);
- The CUDA port methodology, three consensus-relevant bugs surfaced by the differential-oracle property of cross-implementation parity, and what 90.87% line / 100% function LLVM source-based coverage on the spec witness contributes to the production-readiness argument (§11);
- Empirical results across all four backends on the workloads above (§12).

2 Chain-Level Conflict Rules

Every VM in the Lux fleet exposes a single predicate, $\mathsf{conflicts}(v_1, v_2) \in \{\mathsf{true}, \mathsf{false}\}$, over pairs of vertices. The predicate must be *symmetric* and *computable from the vertex batch alone.* Table 1 summarises the per-chain rule.

We say a pair of vertices *commutes* iff it does not conflict.

Chain	Symbol	Engine	Conflict key set	Source
C-Chain (EVM)	C	DAG	$R(v) \cup W(v) \subseteq \mathsf{StorageKey}$	Block-STM
D-Chain (DEX)	D	DAG	$\{(base, quote, side, id)\} \subseteq \mathsf{OrderKey}$	orderbook
X-Chain (UTXO)	X	DAG	$\{\mathsf{txo}_i\} \subseteq \mathsf{UTXO}$	UTXO inputs
Z-Chain (ZK-UTXO)	Z	DAG	$\{\mathsf{null}_i\} \subseteq \mathsf{Nullifier}$	nullifier reveal
Q-Chain (PQ)	Q	DAG+Quasar	$\{\mathsf{stamp}_i\}$	PQ attestation
A-Chain (AI)	A	DAG	$\{\mathsf{jobID}_i\}$	job graph
O-Chain (Oracle)	O	DAG	$\{(\mathsf{feed}, \mathsf{round})\}$	feed rounds
R-Chain (Relay)	R	DAG	$\{(\mathsf{dstChain}, \mathsf{nonce})\}$	relay nonce
S-Chain (ServiceNode)	S	DAG	$\{(\mathsf{node}, \mathsf{epoch})\}$	service epoch
P-Chain (Platform)	P	Linear	ordered	validator set
B-Chain (Bridge)	B	Linear	ordered	ceremony steps
T-Chain (Threshold)	T	Linear	ordered	FROST rounds
M-Chain (MPC)	M	Sharded-linear	per-wallet linear	CGGMP21 rounds
K-Chain (Keys)	K	Linear	ordered	key rotation
I-Chain (Identity)	I	Linear	ordered	identity updates

Table 1: Per-chain conflict rule. The first block operates under the DAG engine with native parallel finality. The second block retains linear ordering where the state machine requires total order (validator registry, cryptographic ceremonies, identity updates).

Definition 1 (Conflict DAG)**.** A *conflict DAG* $D = (V, E, \mathsf{conflicts})$ is a directed acyclic graph whose vertices are VM-specific transaction batches, whose edges are causal parent relationships, and whose conflict predicate satisfies the *Siblings-Commute invariant*: for every pair of vertices v_1, v_2 with $v_1 \not\preceq v_2$ and $v_2 \not\preceq v_1$ in the DAG partial order, $\mathsf{conflicts}(v_1, v_2)$ is false.

The Siblings-Commute invariant is enforced constructively by the VM's BuildVertex: when forming a new vertex the builder scans the current *frontier* (set of rank-maximal accepted vertices) and refuses any batch whose conflict key set overlaps a frontier sibling. This is an $O(|\text{frontier}| \cdot |\text{keys}|)$ bitmap intersection; on GPU it is a single threadgroup-reduce kernel.

3 DAG-EVM

3.1 Lineage: Block-STM Lifted

Block-STM [1] (Aptos Labs, 2022) is an optimistic concurrency control (OCC) algorithm that speculatively executes a linear block's transactions in parallel on separate copies of state, tracks per-transaction read and write sets, and re-executes any transaction whose reads were invalidated by an earlier-committed transaction's writes.

Block-STM's correctness reduces to two properties:

1. Every execution is *conflict-equivalent* to the sequential execution of the same block in the block's canonical order.

2. Every committed write is visible to every strictly-later transaction that reads the same storage location.

These properties are purely local to the block's set of transactions; they make no reference to the fact that blocks are arranged in a linear chain. This is the opening we exploit.

3.2 Construction

Let T be a sequence of pending EVM transactions. The DAG-EVM builder performs:

Algorithm 1 BuildVertex (DAG-EVM)

1: Drain up to N txs from the mempool into T
2: Run Block-STM speculative exec on T over the current state snapshot
3: Emit $(R, W) \leftarrow$ union of per-tx read/write sets
4: $P \leftarrow$ minimal antichain of frontier vertices whose W covers R
5: Let $v.\mathsf{parents} \leftarrow P$
6: $v.\mathsf{id} \leftarrow \mathsf{keccak256}(\mathsf{txHashes} \parallel R \parallel W)$
7: Emit v to the DAG engine

The DAG engine (nebula) votes on acceptance. Once an antichain at rank r is fully populated with accepted vertices, Quasar stamps it with a post-quantum triple-signature certificate and r becomes final. The EVM state is updated by applying the vertices of each finalised antichain in any valid topological order.

3.3 Correctness Sketch

Theorem 2 (Non-Conflict Commutativity, Lean: `nonConflict_commute`)**.** *For any two EVM vertices v_1, v_2 with* $\mathsf{conflicts}(v_1, v_2) = \mathsf{false}$ *and any state σ:*

$$v_2.\mathsf{apply}(v_1.\mathsf{apply}(\sigma)) = v_1.\mathsf{apply}(v_2.\mathsf{apply}(\sigma)).$$

Proof sketch. By case analysis on membership of each storage key k in the write sets of v_1 and v_2. The non-conflict hypothesis excludes write-write overlap; the footprint lemmas (`writesOnly`, `readsOnly`) in the Lean development handle the remaining four cases. Full proof in `Consensus/DAGEVM.lean:60--1`
□

Theorem 3 (Topological Order Invariance, Lean: `topo_equivalence`)**.** *Given the Siblings-Commute invariant, any two topological orderings L_1, L_2 of the same accepted DAG D satisfy* $\mathsf{applyList}(L_1, \sigma) = \mathsf{applyList}(L_2, \sigma)$ *for every initial state σ.*

Theorem 4 (No Double Spend, Lean: `no_double_spend`)**.** *If v, w are both accepted, $v \neq w$, and both write the same storage key k, then v and w must be in distinct ranks (one is an ancestor of the other in the DAG).*

Corollary 5 (Serialisability)**.** *Every DAG-EVM execution is conflict-equivalent to a sequential EVM execution over the same set of transactions. An external observer sees a linear history of state transitions; the underlying parallel vertex topology is invisible at the state projection.*

4 DAG-Z-Chain

The Z-Chain operates the classical ZK-UTXO model: each transaction reveals a set of nullifiers (the anonymous counterparts of spent UTXOs), creates new commitments, and carries a ZK proof (Groth16 or Plonk) that the nullifier/commitment relationship is valid under a known verification key. The Z-state is the pair $(\mathsf{spent}, \mathsf{committed})$ with both components monotone (never shrink).

The conflict rule is dramatically simpler than in the EVM:

$$\mathsf{conflicts}(v_1, v_2) \iff N(v_1) \cap N(v_2) \neq \emptyset$$

where $N(v)$ is the union of nullifier sets across the transactions of v.

Theorem 6 (Z-Chain No Double Spend, Lean: `no_double_spend`)**.** *No nullifier appears in two distinct accepted vertices of the Z-Chain DAG.*

Theorem 7 (Batch Verification Soundness, axiom: `batch_sound`)**.** *Groth16 (and Plonk) batch verification is sound iff each constituent proof is individually sound. GPU-accelerated batch verify preserves this property under the pairing-based (resp. polynomial-commitment) assumption.*

Theorem 8 (FHE Commutativity, Lean: `fhe_linearity_preserved`)**.** *If two Z-Chain vertices each perform an additive FHE balance update on disjoint nullifier keys, the two updates commute.*

These together give DAG-Z-Chain the same parallelism profile as DAG-EVM while preserving all of the ZK-UTXO security invariants, *without* additional cryptographic assumptions beyond those already underlying the proof system.

5 Lean 4 Development

The full development is in `lux/formal/lean/Consensus/`:

- `DAGEVM.lean` — DAG-EVM safety, commutativity, topological order invariance, no-double-spend, serialisability.
- `DAGZChain.lean` — DAG-Z-Chain nullifier-disjointness, proofs-commute, parallel-verify soundness, FHE linearity.
- `PQZParallel.lean` — cross-chain non-interference: running C/X/Z/Q/D/A/O/R/S DAGs concurrently preserves per-chain safety; Quasar stamps are independently verifiable; validator signing across k chains scales linearly.
- `Safety.lean`, `Liveness.lean`, `BFT.lean` — Pre-existing per-chain consensus safety (not modified).
- `Quasar.lean` — PQ certificate unforgeability (not modified); supplies the finality assumption for DAG antichains.

Build: `cd lux/formal/lean && lake build`.

6 TLA$^+$ Specifications

The TLA$^+$ models live in `lux/formal/tla/`:

- `DAGEVM.tla` + `MC_DAGEVM.cfg` — state variables accepted, finalRank, mempool; actions Propose, Finalize; invariants `TypeOK`, `Safety`, `NoDoubleSpend`, `Serializability`, `ConflictsAlongEdges`; property `Liveness`.
- `DAGZChain.tla` + `MC_DAGZChain.cfg` — state variables accepted, spent, committed, finalRank; invariants `NoDoubleSpend`, `AntichainDisjoint`.

TLC model-checking runs against small instances ($|\mathsf{Keys}| = 3, |\mathsf{Vertices}| = 6$) and enumerates the full reachable state space in bounded time.

System	Exec parallelism	Consensus DAG?	Conflict source	State model
Ethereum	none	linear	–	EVM (account)
Aptos	Block-STM	linear	r/w sets	Move (resource)
Sui (Mysticeti)	object-type	DAG (Narwhal)	object IDs	Move (object)
Solana	account-lock	linear	static lockset	account
Monad	OCC + pipeline	linear	r/w sets (post-hoc)	EVM
Fuel	UTXO-parallel	linear	UTXO inputs	UTXO
Aleph Zero	none	DAG (AlephBFT)	–	substrate
Lux C-Chain (DAG-EVM)	Block-STM	**DAG**	**r/w sets**	EVM
Lux X-Chain	DAG-native	**DAG**	UTXO inputs	UTXO
Lux Z-Chain (DAG-Z)	parallel verify	**DAG**	nullifiers	ZK-UTXO
Lux Q-Chain	attest-parallel	**DAG+Quasar**	PQ stamps	attestation

Table 2: Lux versus contemporary parallel / DAG blockchains. Lux is the only production system in which (a) the execution engine's conflict graph is the consensus DAG and (b) this shape holds uniformly across EVM, UTXO, ZK-UTXO, and PQ attestation workloads.

7 Comparison with Contemporaries

The critical distinction: Sui's Narwhal DAG carries *transaction availability*, not the execution conflict graph. The DAG is collapsed to a total order by Bullshark/Mysticeti and then executed sequentially (or parallel by object-type, separately). Aptos' Block-STM operates inside a linear block: if block B_n is not yet finalised, no transaction of B_{n+1} can execute even if entirely independent. Monad stacks pipelining on linear consensus; DAG-level parallelism is not available. Solana's account-lock model is static (declared in the tx), not derived from execution footprints, and does not compose with DAG consensus.

Lux is the only production design in which the same conflict predicate drives intra-block execution (Block-STM) *and* inter-block finality (nebula DAG) — yielding a uniform model across all chains with a natural parallelism axis (EVM, DEX, UTXO, ZK, Oracle, Relay).

8 GPU-Native EVM Execution

The critical path of DAG-EVM is:

1. **Speculative Block-STM execution** of candidate vertex's txs.
2. **ecrecover** for every transaction in the batch.
3. **Conflict scan** against the frontier's bitmap union.
4. **keccak** for vertex ID and inclusion in the next level's Merkle-Patricia state root.

Through the v0.20.0 → v0.23.0 release cycle (luxcpp/evm tags `v0.22.0`, `v0.23.0`), the GPU stage stopped being a single ecrecover kernel and grew into a full opcode-level interpreter for both Apple Metal and NVIDIA CUDA, sharing one host-side dispatcher. The result is that every step above can run on the same GPU device.

8.1 Kernel Surface

The GPU EVM kernel is a single-source artefact in two dialects:

- `lib/evm/gpu/kernel/evm_kernel.metal` (1067 SLOC, Metal Shading Language 3.0).
- `lib/evm/gpu/cuda/evm_kernel.cu` (1713 SLOC, CUDA C++ for sm_80 / sm_90).

Both implement the Cancun-era opcode set including `KECCAK256`, the Cancun context family (`ORIGIN`/`GASPRICE`/`BLOCKHASH`/`CHAINID`/`BASEFEE`/`BLOBHASH`/`BLOBBASEFEE`), `MCOPY`, `TLOAD`/`TSTORE`, and `LOG0`–`LOG4`. ABI version 3 of the kernel adds a `BlockContext` buffer plus transient-storage and log buffers (Metal binds them at indices 8–12). Account-state opcodes (`BALANCE`, `EXTCODESIZE`, ...) return safe defaults inside the kernel; live host data is provided through the bridge described in §8.3.

8.2 Precompile Dispatch

The precompile dispatcher routes addresses `0x01`–`0x11` through GPU code paths where it is profitable. `ECRECOVER` (`0x01`) executes on Metal and CUDA via `secp256k1_recover.cu`; the BLS12-381 (`0x0b`–`0x11`) EIP-2537 family is wired through the same dispatcher. The standalone unit target `evm-precompile-test` reports 19 precompile groups passing on the M1 Max reference hardware; the run is guarded by `available_backends()` so a host without a Metal device falls back to CPU silently.

8.3 EVMC Host Bridge for CALL/CREATE

The headline limitation of a pure-GPU EVM is that the kernel cannot make a runtime callback into an EVMC `Host` for nested `CALL`/`CREATE` frames; GPUs do not call back to CPU memory mid-warp. The v0.22.0 host bridge (`lib/evm/gpu/host/evmc_bridge.cpp`) sidesteps this by performing a static analysis on the entrypoint bytecode: it walks the basic-block graph, identifies every `CALL`/`CREATE` target whose address is a constant immediate, pre-resolves the destination code from the supplied `evmc::Host`, and packs the entire reachable code closure into a single GPU-side *code dictionary*. Dynamic-target calls (`CALL` where the address is computed at runtime) are rejected and the transaction falls back to evmone on CPU.

The accept/reject decisions and depth enforcement are exercised by `test/unittests/host_bridge_test.cpp`, with the seven scenarios listed in Table 3. The bridge enforces a maximum frame depth of 4; deeper chains route to CPU.

Scenario	**Outcome**
Pure leaf (no CALL/CREATE)	GPU accept
Static `CALL` with const target	GPU accept (pre-resolved)
Dynamic `CALL`	CPU fallback
Deterministic `CREATE`	GPU accept (deterministic addr)
4-deep static `CALL` chain	GPU accept
9-deep static `CALL` chain	CPU fallback (depth > MAX)
`DELEGATECALL` context	GPU accept; preserves caller

Table 3: Host-bridge accept/reject scenarios. All seven cases are checked byte-for-byte against the same transaction routed through evmone on CPU in `test/unittests/host_bridge_test.cpp`.

8.4 Unified Pipeline

A single Metal device drives EVM execution, consensus state hashing (`lib/evm/gpu/gpu_state_hasher.hpp`), and post-quantum signature verification (BLS12-381 + ML-DSA-65 hosted by `lib/evm/gpu/metal/bls_host.mm`). The pipelined run measured at v0.22.0 on M1 Max reports a 1.07× speedup vs serial (consensus + EVM + sig-verify) at 19 blocks/s sustained over 1000 blocks — the modest factor reflects the fact that each stage already saturates a different GPU resource (compute for EVM, memory bandwidth for keccak state hashing, BLS scalar-mult for sig-verify), so pipelining recovers only the overlap. Source: `lib/evm/gpu/test_unified_pipeline.cpp`.

8.5 Empirical Speedup, M1 Max

Two regimes need separate accounting: (a) the original ecrecover/keccak-dominated bottleneck that motivated the GPU port at v0.20.0, and (b) the bytecode-level kernel that came online at v0.22.0. Table 4 shows the ecrecover/keccak path is essentially solved (the original 32× ecrecover speedup carries over) and that the bytecode kernel itself is faster than evmone on per-opcode microkernels once the batch is large enough to amortise GPU dispatch.

Stage	**CPU evmone**	**GPU Metal**	**Speedup**
ecrecover (47K sigs)	1613 ms	50 ms	32.3×
keccak256 (47K hashes)	127 ms	18 ms	7.1×
Bytecode kernel (`evm-bench-kernel`, ADD-loop tx)			
GPU V1 (1 thread/tx)	—	252 M ops/s	1.62×
CPU evmone reference	155 M ops/s	—	1.00×
Real ADD-loop tx (50 iter, N=2048)			
GPU V1	—	25,327 TPS	—
		1.14 M EVM ops/s	—

Table 4: Measured GPU EVM speedups on Apple M1 Max, 32-core Metal GPU. ecrecover and keccak figures reproduce in `lib/evm/gpu/metal/bench_keccak.cpp`. Bytecode-kernel figures from `lib/evm/gpu/kernel/bench_kernel.cpp`; gas-match against CPU evmone is asserted on every run (the bench fails closed if a single tx's `gas_used` disagrees). The GPU kernel does not yet beat the Block-STM CPU-Parallel path on every microkernel — `results/speedup-with-buffer-cache-2026-02-28.txt` catalogs the $N \leq 2048$ regimes where CPU-Parallel still wins; that is the workload where a per-thread CPU evmone interpreter outpaces a one-tx-per-warp GPU launch. The bytecode kernel wins above this batch size and on workloads where ecrecover dominates.

GPU is auto-detected at runtime: if the dispatcher's `available_backends()` returns a non-CPU backend, the bytecode is routed through the GPU host launcher (`evm_kernel_host.mm` on Apple, `cuda/evm_kernel_host.cpp` on NVIDIA). When no GPU is present, both CPU paths (`CPU_Sequential` and `CPU_Parallel` via Block-STM) run unchanged. There is no user flag.

8.6 Limitations

- *Dynamic CALL targets fall back to CPU.* The host-bridge static analysis only resolves constant-immediate targets. Solidity code that loads a callee address from storage at runtime is not yet GPU-eligible. Path: extend the dictionary to include the closure of all addresses ever stored in dispatcher slots, with re-staging on dictionary miss. Not yet implemented.

- *SELFDESTRUCT is rejected.* The kernel returns `CallNotSupported` on `0xFF`; the dispatcher falls back to CPU. Acceptable today because Cancun deprecates the opcode for new accounts.
- *Account-state opcodes return defaults inside the kernel.* `BALANCE`, `EXTCODESIZE`, `EXTCODEHASH`, `EXTCODECOPY`, `SELFBALANCE` read from a host-supplied snapshot dictionary; addresses missing from the dictionary surface as zero. The host bridge populates the dictionary for every address it has pre-resolved; addresses outside that closure read zero. The parity test corpus catches this with the `state_*` vectors.
- $N \leq 2048$ *batches lose to CPU-Parallel.* GPU dispatch cost (host → device buffer staging + threadgroup launch) is in the 6 ms range on M1 Max for cold buffers, dropping to ∼4 ms once buffer cache is warm. CPU-Parallel Block-STM clears the same workload in ∼0.1 ms–4 ms. The crossover moves into the kernel's favour at $N \geq 4096$ (workload-dependent), or when the workload contains ecrecover/keccak which still dominate on CPU.

9 Plugging the GPU EVM into Quasar Consensus

The previous section established that a single Metal/CUDA kernel can execute a Cancun-era block end-to-end. This section answers a separate question: *where* in the Quasar consensus pipeline does that execution actually get invoked, against *which* interface, and at *what* batch sizes does the GPU win.

9.1 What Quasar Decomposes Into

The consensus repository (`github.com/luxfi/consensus` at SHA `5bfc6ac`) is a pure-protocol library: it contains no execution logic and no state model. It exposes two engine modes wired from the same sub-protocols:

- **Linear chain** (`protocol/nova`, driven by `engine/chain`) — Photon → Wave → Focus → Ray → Sink. Used by the C-Chain.
- **DAG** (`protocol/nebula`, driven by `engine/dag` and `protocol/field`) — Photon → Wave (per frontier vertex) → Flare (cert/skip) → Horizon (safe-prefix scan) → Field (commit). This is the path described in §8 for the DAG-EVM.

The components, with file references in `~/work/lux/consensus`:

- *DAG vertex production.* `protocol/field/field.go:30` defines `Proposer[V].Propose(ctx, parents) (V, error)`. The *networking* layer pushes a payload into the proposer; the proposer returns a vertex ID and ships it. No bytecode runs here — a vertex is just `(parents, author, round, payload bytes)`.
- *Wave-based ordering.* `protocol/wave/wave.go` (the `Wave[T]` struct) drives threshold voting per frontier vertex. The FPC selector (`protocol/wave/fpc/`) randomises the threshold $\theta \in [\theta_{\min}, \theta_{\max}]$ from a PRF seed to defeat adaptive adversaries. Wave alone returns `(preferOK, confOK)` per round; it does not commit.
- *FPC final commitment.* `protocol/field/field.go:107 Driver[V].Tick` computes a *safe prefix* via `computeSafePrefix` (line 119), then calls `Committer[V].Commit(ctx, ordered)`. The committer's `Commit` call is the post-FPC hook: every vertex passed in is finalised with all its causal ancestors finalised.

- *Quasar certificate.* `protocol/quasar/types.go:40` `QuasarCert` carries the BLS aggregate, Pulsar share, and Z-Chain Groth16-rolled-up ML-DSA proof. The cert is produced *after* `Commit` returns; it certifies the safe prefix, not individual transactions.

9.2 Where Bytecode Execution Lives

The crucial point: *Quasar does not call into the EVM.* Quasar orders opaque vertex IDs. The EVM call lives one layer above Quasar, in the VM that hosts the chain.

Linear chain (Nova). `engine/chain/block/block.go:88--130` defines `ChainVM`, which the engine drives. The engine's call sequence (`engine/chain/engine.go:947--1019`, function `buildBlocksLocked`) is:

1. `vm.BuildBlock(ctx)` returns a `block.Block` (an interface, not bytes; line 94 of `block.go`).
2. *Engine releases its mutex* (line 976) and calls `vmBlock.Verify(ctx)` (line 977). **This is the EVM execution hook.** `Verify` runs the bytecode and returns the post-state. If `Verify` fails, the block is dropped and never enters consensus.
3. Engine adds the block to consensus (`t.consensus.AddBlock`, line 983) and self-votes.
4. On finalisation: `vmBlock.Accept(ctx)` (line 1000 for K=1, line 877 for the multi-validator finaliser `tryFinalizeBlock`, and line 924 for the engine's `DrainAccepted` loop).

So bytecode runs *during vertex production*, before consensus voting. `Accept` is a state-commitment hook — it persists what `Verify` already computed, it does not re-execute.

DAG mode (Nebula). The same split shifts one level: `Proposer.Propose` runs the bytecode for the proposed vertex's transactions and returns a vertex ID that already commits to the post-state root. `Field.Commit` then delivers an *ordered* list of vertex IDs whose post-state was computed at proposal time. The committer's job is to *apply* those state diffs against the now-canonical history — not to re-run the EVM.

This split is forced by the protocol shape: Quasar votes on vertex IDs, and a vertex's ID must commit to its post-state for the cert to be meaningful. We therefore execute speculatively at proposal, and apply deterministically at commit. The Block-STM scheduler in `cevm` (`lib/evm/gpu/parallel_engine.cpp`) provides the speculative re-execution-on-conflict that this requires.

9.3 The Interface Quasar Calls

The full `ChainVM` surface area Quasar's chain engine drives, from `engine/chain/block/block.go:88--130` (verbatim):

```
type ChainVM interface {
    Initialize(ctx context.Context, init Init) error
    BuildBlock(context.Context) (Block, error)
    ParseBlock(context.Context, []byte) (Block, error)
    GetBlock(context.Context, ids.ID) (Block, error)
    SetPreference(context.Context, ids.ID) error
    LastAccepted(context.Context) (ids.ID, error)
    GetBlockIDAtHeight(context.Context, uint64) (ids.ID, error)
    SetState(context.Context, uint32) error
```

```
    Shutdown(context.Context) error
    Version(context.Context) (string, error)
    Connected(context.Context, ids.NodeID, *version.Application) error
    Disconnected(context.Context, ids.NodeID) error
    HealthCheck(context.Context) (HealthCheckResult, error)
    NewHTTPHandler(context.Context) (http.Handler, error)
    WaitForEvent(context.Context) (Message, error)
}

type Block interface {
    ID() ids.ID
    Parent() ids.ID
    ParentID() ids.ID
    Height() uint64
    Timestamp() time.Time
    Status() uint8
    Verify(context.Context) error   // <-- bytecode here
    Accept(context.Context) error   // <-- commit state here
    Reject(context.Context) error
    Bytes() []byte
}
```

For the DAG path, `engine/dag/vertex/vertex.go:11--28` adds:

```
type Vertex interface {
    ID() ids.ID
    Bytes() []byte
    Height() uint64
    Epoch() uint32
    Parents() []ids.ID
    Txs() []ids.ID
    Status() choices.Status
    Verify(context.Context) error
    Accept(context.Context) error
    Reject(context.Context) error
}
type DAGVM interface {
    ParseVertex(context.Context, []byte) (Vertex, error)
    BuildVertex(context.Context) (Vertex, error)
}
```

Two hooks, both for the GPU EVM: `Verify` for execution, `Accept` for state commit.

9.4 Concrete Integration Glue

The `cevm` package (`~/work/lux/cevm/cevm.go` at SHA `4d652b2`) already exports the C ABI:

```
func ExecuteBlockV2(backend Backend, numThreads uint32,
    txs []Transaction) (*BlockResultV2, error)
```

The integration component is a Go struct that satisfies `block.Block` and routes `Verify` into `cevm.ExecuteBlockV2`. The plumbing is direct because the consensus library already keeps execution and ordering decoupled — the only state that flows from the cevm result into the consensus layer is the 32-byte state root (carried in `Block.Bytes()`) and the boolean `Verify` return.

```
// cevmBlock satisfies engine/chain/block.Block and engine/dag/vertex.Vertex.
type cevmBlock struct {
    id        ids.ID
    parent    ids.ID
    height    uint64
```

```
    ts         time.Time
    txs        []cevm.Transaction
    backend    cevm.Backend
    threads    uint32
    // populated by Verify, read by Accept:
    result     *cevm.BlockResultV2
    stateRoot  [32]byte
    raw        []byte
    state      uint8 // choices.Processing / Accepted / Rejected
    parent2    *cevmBlock // for state diff application at Accept
    db         kvWriter
}

func (b *cevmBlock) ID() ids.ID         { return b.id }
func (b *cevmBlock) Parent() ids.ID     { return b.parent }
func (b *cevmBlock) ParentID() ids.ID   { return b.parent }
func (b *cevmBlock) Height() uint64     { return b.height }
func (b *cevmBlock) Timestamp() time.Time { return b.ts }
func (b *cevmBlock) Status() uint8      { return b.state }
func (b *cevmBlock) Bytes() []byte      { return b.raw }

// Verify is the GPU EVM hook. Called by engine/chain/engine.go:977.
func (b *cevmBlock) Verify(ctx context.Context) error {
    if b.result != nil {
        return nil // memoised
    }
    res, err := cevm.ExecuteBlockV2(b.backend, b.threads, b.txs)
    if err != nil {
        return fmt.Errorf("cevm␣verify␣height=%d:␣%w", b.height, err)
    }
    // Cross-check vs the proposer's claimed root in raw block bytes.
    if !bytes.Equal(res.StateRoot[:], b.expectedRoot()) {
        return errStateRootMismatch
    }
    // Reject any tx with TxError; TxRevert is a valid in-protocol failure.
    for i, s := range res.Status {
        if s == cevm.TxError {
            return fmt.Errorf("tx␣%d␣returned␣TxError", i)
        }
    }
    b.result = res
    copy(b.stateRoot[:], res.StateRoot[:])
    return nil
}

// Accept is the state-commit hook. Called by engine/chain/engine.go:877,
// engine.go:924 (DrainAccepted), or engine.go:1000 (K=1 fast path).
func (b *cevmBlock) Accept(ctx context.Context) error {
    if b.result == nil {
        return errVerifyNotRun
    }
    // Persist the diff produced by Verify. The cevm bridge surfaces gas
    // and status; the underlying state diff is pulled from the host
    // bridge's account-state dictionary.
    if err := b.db.applyDiff(b.result, b.stateRoot); err != nil {
        return err
    }
    b.state = choicesAccepted
    return nil
```

```
}

func (b *cevmBlock) Reject(ctx context.Context) error {
    b.state = choicesRejected
    b.result = nil // free GPU result
    return nil
}
```

The VM-level `ChainVM.BuildBlock` constructs a `cevmBlock` from the mempool, picks the backend (`cevm.AutoDetect()` at boot, `cevm.Health()` verified), and returns it. The engine then drives `Verify`, votes, and on quorum calls `Accept`. No code in `consensus/` changes; the GPU EVM is a pure plug into `block.ChainVM`.

9.5 Where the Two Parallelisms Meet

Quasar can finalise multiple DAG vertices concurrently — specifically, any antichain of the conflict-free DAG (§**??**). The GPU EVM can execute thousands of transactions concurrently within one launch. These are *orthogonal* dimensions:

Source	**Granularity**	**Limit**
Quasar DAG width	vertices per round	antichain size of DAG
GPU EVM batch size	txs per dispatch	GPU SIMD lane count

The dimension that dominates is the one with the lower throughput floor. On M1 Max the GPU launch cost is 6 ms cold, 4 ms warm (§8.5); the per-vertex proposer cost on the consensus side is dominated by the wave round-trip (250 ms default `RoundTO`, `protocol/field/field.go:67`). *Consensus dominates by ∼40×.* The GPU has plenty of slack to amortise launch cost across a vertex's transactions, but it cannot make the wave round shorter.

Pipelining helps when the GPU launch on vertex V_n overlaps with the wave round on V_{n-1}. In the linear chain mode, the engine signals `t.signalPipeline()` after every accept (`engine.go:894`), so a build/verify can begin while the previous block is still being gossiped. Pipelining hurts when the proposer must wait for the GPU launch to drain before issuing a fresh `Verify` — the cevm `ExecuteBlockV2` is currently synchronous (the C bridge blocks on `[commandBuffer waitUntilCompleted]`). Async dispatch is listed as an open item in `lib/evm/gpu/PERF.md` (priority 2).

9.6 Saturation Math

The required regime, given the cevm dispatch crossover plot (`lib/evm/gpu/PERF.md`, "Crossover analysis"):

$$
\begin{aligned}
T_{\text{launch}}^{\text{cold}} &= 6\,\text{ms} \\
T_{\text{launch}}^{\text{warm}} &= 4\,\text{ms} \\
T_{\text{op}}^{\text{GPU}} &= \frac{1}{339\text{M ops/s}} \approx 2.95\,\text{ns/op} \\
T_{\text{op}}^{\text{CPU}} &= \frac{1}{159\text{M ops/s}} \approx 6.29\,\text{ns/op}
\end{aligned}
$$

Break-even (warm) for a generic workload of W opcodes per dispatch:

$$T_{\text{launch}}^{\text{warm}} + W \cdot T_{\text{op}}^{\text{GPU}} < W \cdot T_{\text{op}}^{\text{CPU}} \quad \Rightarrow \quad W > \frac{T_{\text{launch}}^{\text{warm}}}{T_{\text{op}}^{\text{CPU}} - T_{\text{op}}^{\text{GPU}}} \approx \frac{4\,\text{ms}}{3.34\,\text{ns}} \approx 1.2 \times 10^6 \text{ opcodes.}$$

So a single dispatch needs ∼1.2M opcodes for the GPU to win strictly on bytecode. The 100 K txs/sec target translates as follows. A vertex of 100 txs at 50 EVM ops/tx delivers 5000 ops per dispatch — well *below* the break-even. A vertex of 100 txs at 600 ops/tx (loop-heavy contracts) delivers 60,000 ops — still below. The break-even at 1.2M ops requires either:

1. *Fat workload per dispatch:* 100 txs × 12,000 ops/tx (heavy contract — DEX swap, AMM compound), or
2. *Bigger batch:* 12,000 txs at 100 ops/tx, or
3. *Pipelined dispatch* that amortises launch across multiple blocks.

The 100 K txs/sec (1000 vertices × 100 txs) *aggregate* is more than enough — if 1000 vertices/sec is emitted from a wave of 5–10 parallel proposers, each proposer dispatches to a separate cevm host (`cevm.ExecuteBlockV2` is goroutine-safe per `cevm.go:17`, with thread-local kernel hosts). Aggregating across proposers gives 200–1000 txs per dispatch on average, still below break-even at the 50-op-per-tx floor.

The ecrecover and keccak paths are different. Their break-even sits much lower because the per-op CPU cost is high (34 µs/sig for ecrecover — Table 4). At 47,000 signatures per dispatch the GPU runs in 50 ms versus the CPU's 1613 ms. A 100-tx block with two signatures per tx (200 sigs) is below crypto break-even too; a 10,000-tx block (or 50 blocks of 200 sigs each batched together) is firmly above it. *In production, signature verification is the path that justifies the GPU; opcode execution wins only on fat workloads or via pipelining.*

9.7 Limitations

- *No host callback at vertex boundary.* `cevm.ExecuteBlockV2` is a single C call that returns when the full batch finishes. The GPU kernel cannot ask Quasar mid-dispatch "did this vertex finalise yet?" because there is no host-callback primitive in the bridge. Consequence: the EVM cannot observe consensus state changes (e.g., a finalised cross-shard read) within a single `Verify`. The host bridge handles `CALL`/`CREATE` only via static analysis on the entry bytecode (§8.3); cross-vertex callbacks would require either re-issuing a fresh dispatch or a CPU fallback path.
- *Synchronous dispatch.* `[commandBuffer waitUntilCompleted]` blocks the calling goroutine. While the consensus engine has a per-block goroutine pool (`engine.go buildBlocksLocked`), a backed-up GPU draining a fat dispatch will pile up subsequent `Verify` calls. Async dispatch (priority 2 in PERF.md) is required for true pipelining across vertices.
- *Account-state opcodes return defaults.* `BALANCE`, `EXTCODESIZE`, `EXTCODEHASH`, `EXTCODECOPY`, `SELFBALANCE` read a host-supplied snapshot dictionary. For a vertex that touches an address outside the static-analysis closure, the kernel reads zero. The integration glue (`cevmBlock.Verify`) does not detect this and would accept a wrong-answer block; the parity-test corpus (§10) is the only catch. A *stricter* integration would re-run on CPU when any opcode misses the dictionary — not yet wired.

- *Dynamic-target `CALL` forces CPU fallback.* Per §8.3, a runtime-computed callee address routes the entire transaction to evmone on CPU. This is correct but breaks the GPU-batch invariant: a block of 100 txs in which one tx has a dynamic CALL splits into one CPU run plus a 99-tx GPU run, doubling the total wall-clock for that block.

- *Quasar does not vote on state roots, only vertex IDs.* A vertex's payload bytes commit to the post-state root, but Quasar itself never re-checks that root. If two honest validators execute the same vertex on different cevm backends (e.g., one Metal, one CPU-Parallel) and produce different roots due to a parity bug, only the four-way parity test corpus (`test/parity_test.cpp`) catches it before deployment. Run-time disagreement would surface as a vote split, not a halt — so divergent backends would silently lose finality on the affected branch. The fix path is exactly the parity corpus discipline: every backend must be byte-equivalent on every consensus-critical vector.

- *No state-streaming for huge dispatches.* The cevm result is materialised entirely in host RAM (the V2 ABI returns a single `state_root[32]` plus per-tx arrays). A 100 K-tx dispatch that needed to stream state diffs to a database during execution cannot be expressed in the current ABI. Block-STM provides the speculation; persistence is bulk-applied at `Accept`. For bytecode that mutates millions of slots, this means the GPU result sits in RAM until the consensus quorum lands, which is acceptable at today's chain TPS but will need a streaming path before a single GPU dispatch routinely exceeds available host RAM.

Net. The integration is small — a `block.Block` adapter, one C call per `Verify`, no consensus changes — because the consensus library is intentionally execution-free. The hard work is on the GPU side (parity, host bridge, V2 kernel for SIMD-cooperative dispatch). The right way to read this section is that the seam is already correct; what remains is closing the break-even gap so that GPU dispatch wins at vertex sizes the network actually emits.

10 Four-Way Parity as a Production Argument

The claim of consensus-equivalent execution across four backends is not a slogan; it is a single test driver that compiles, links, and runs all four in the same process.

10.1 The Four Backends

`evm::gpu::Backend` enumerates four routes, each invoked through a shared dispatcher entrypoint `execute_block(config, txs, nullptr)`:

1. `CPU_Sequential` — the kernel CPU interpreter (`lib/evm/gpu/kernel/evm_interpreter.hpp`, header-only), single-threaded, opcode-for-opcode reflection of the Metal kernel. This is the *spec witness*: it shares the source of truth with the GPU kernels.

2. `CPU_Parallel` — the same interpreter run by the Block-STM scheduler with N worker threads (`lib/evm/gpu/parallel_engine.cpp`).

3. `GPU_Metal` — the Apple Metal kernel (`evm_kernel.metal`), one threadgroup per transaction, dispatched by `lib/evm/gpu/kernel/evm_kernel_host.mm`.

4. `GPU_CUDA` — the NVIDIA CUDA kernel (`lib/evm/gpu/cuda/evm_kernel.cu`), one warp per transaction, dispatched by `lib/evm/gpu/cuda/evm_kernel_host.cpp`.

The CUDA arm is gated on the build define `EVM_CUDA`. On macOS the define is absent so the CUDA assertions compile out and only Apple GPU remains in the loop; on Linux/CUDA CI (the AWS EKS workflow described in §11.4) the define flips on and all four backends are checked.

10.2 Vector Corpus

The corpus is built once per process startup (`test/unittests/parity_test.cpp`, lines 92–573). It contains 133 bytecode vectors hand-written to exercise:

- *Arithmetic*: `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `ADDMOD`, `MULMOD`, `EXP`, signed variants;
- *Comparison* and *bitwise*: signed/unsigned `LT`/`GT`/`SLT`/`SGT`, `ISZERO`, `AND`/`OR`/`XOR`/`NOT`, `BYTE`/`SHL`/`SHR`/`SAR`;
- *Hashing*: `KECCAK256` on empty / `abc` / 32-byte-zero / 256-byte-zero inputs;
- *Context*: `ADDRESS`, `ORIGIN`, `CALLER`, `CALLVALUE`, `CHAINID`, `GASPRICE`, `COINBASE`, `TIMESTAMP`, `NUMBER`, `GASLIMIT`, `BASEFEE`, `BLOBHASH`, `BLOBBASEFEE`, `BLOCKHASH`;
- *Calldata*, *code*, *memory* (`MSTORE`/`MLOAD`/`MSTORE8`/`MSIZE`/`MCOPY`);
- *Storage* (`SSTORE`/`SLOAD`, `TSTORE`/`TLOAD`, zero-load);
- *Control flow* (`JUMP`, `JUMPI`, `PC`, `GAS`, `STOP`, `INVALID`, undefined opcode, bad jump);
- *Stack*: `PUSH0` through `PUSH32`, `DUP1`–`DUP16`, `SWAP1`–`SWAP16`, `POP`;
- *Logging* (`LOG0`–`LOG4`);
- *Returndata* (`RETURN`/`REVERT`/empty/word);
- *Account state* (default-zero through the kernel);
- *CALL family* expected to return `CallNotSupported` (`CALL`, `CALLCODE`, `DELEGATECALL`, `STATICCALL`, `CREATE`, `CREATE2`, `SELFDESTRUCT`);
- *Out-of-gas*.

Each vector carries an `Expectation` tag — `Agree`, `GasOnly`, or `KernelCpuMissing` — that tells the test which divergences are tolerated. `Agree` demands bit-exact match on `(gas_used, status, output)` across all four backends. `GasOnly` permits gas-accounting drift (typically the cost of `MSTORE` memory expansion in different revisions) but still requires status and output to match. `KernelCpuMissing` marks vectors where the kernel CPU interpreter has not yet implemented an opcode that the GPU kernels do; these vectors expect CPU `Error` and GPU success — catching the case where a new opcode lands on GPU before the CPU spec witness catches up.

10.3 The Test

`TEST(Parity, AllBackendsAgreeOnEveryVector)` loops the corpus once, runs each vector through all four backends, and counts the `(Agree ok, GasOnly ok, Missing ok)` totals. Any divergence under the active expectation registers as `ADD_FAILURE`; the test asserts at the end that the failure count is zero. The test is a single GoogleTest case so it can run inside `ctest -j` alongside the other unit suites.

The output prints, on every CI run:

```
[parity] runs:  CPU_Sequential=N CPU_Parallel=N GPU_Metal=N [GPU_CUDA=N]
[parity] Agree:            <ok> ok, <fail> fail
[parity] GasOnly:          <ok> ok, <fail> fail
[parity] KernelCpuMissing: <ok> ok, <fail> fail
[parity] total <pass> / <total> vectors meet expectations
```

10.4 Why This Is Not a Smoke Test

Three properties make four-way parity a production-grade argument rather than a developer-confidence test.

(P1) Independent implementations of the same spec. The kernel CPU interpreter and the Metal kernel share a directory and review cycle but no opcode-level code. CUDA was ported from Metal in a separate branch by line-by-line translation (§11) and produces an independent control-flow graph: the Metal version uses `threadgroup` reductions for stack ops, the CUDA version uses warp shuffles. Block-STM adds a fourth code path because the scheduler's read/write tracking is a different layer of the system. A bit-exact result across all four is therefore evidence that the EVM specification was internalised, not that one implementation was copy-pasted.

(P2) Status, gas, and output checked exactly. The default `Expectation::Agree` requires equality on three independent quantities. Status is a discrete enum; gas is a 64-bit integer; output is a byte vector of any length. There are 13 vectors marked `GasOnly` and a small set marked `KernelCpuMissing` — each tagged exception is a documented divergence with a specific algebraic cause, not a mass-pardon. The test writes a `[parity]` promotion hint to `stdout` when a `GasOnly` vector starts agreeing, forcing the engineer either to upgrade the tag or explain the regression.

(P3) The four backends are co-deployed. The same dispatcher `execute_block(config, txs, nullptr)` is what production callers use. There is no separate "test mode" path. A consensus-relevant bug that crept into the Metal kernel would be caught by the CUDA assertion; a CUDA-only bug would fail the Metal assertion; a kernel-CPU-only bug would fail either GPU. The cross-checking is symmetric and runs on every PR through CI.

10.5 What the Coverage Numbers Add

Source-based LLVM coverage is built into the same target set (`lib/evm/COVERAGE.md`). The parity-driver run at v0.23.0 reports **90.87% line coverage and 100% function coverage** on the kernel CPU interpreter (`evm_interpreter.hpp`). The 9.13% of unexecuted lines are precisely the unimplemented `CALL`/`CREATE` arms (which the test catches under `Expectation::Agree` for `CallNotSupported`) and the dead branches behind compile-time `assert`s. The combination — four backends agreeing across 133 hand-crafted vectors plus 100% function coverage on the spec witness — means we have a counter-example finder if any backend silently diverges from the others on an opcode the corpus already touches.

10.6 Limitations

- *Coverage on `.metal` and `.cu` sources is structural only.* LLVM source-based coverage instruments the kernel-CPU interpreter (which is a `.hpp`) and the host launchers (`.mm`/`.cpp`). Metal AIR and CUDA PTX are outside LLVM's coverage scope (Metal compiles to AIR, nvcc emits PTX without coverage-mapped IR). Coverage on the GPU code itself is indirect: every vector that executes through the GPU dispatcher necessarily executes the corresponding device path. The four-way parity test is the corroboration.

- *The corpus is per-opcode, not stateful.* Each vector runs in its own fresh state. A consensus bug that requires multiple transactions to manifest (a Block-STM hazard, an SSTORE refund accumulator inconsistency across txs) is not currently in the corpus. The Block-STM scheduler has its own unit suite (`lib/evm/gpu/parallel_engine.cpp` drives `evm-test-modes`) but it does not cross-validate against the other three backends. Path: extend `ParityVector` to a `ParitySequence` type carrying multiple `Transaction`s and a state-checkpoint diff. Not yet implemented.
- *No fuzzing layer.* The corpus is hand-written and grows by inspection. A differential fuzzer that emits random bytecode and checks four-way agreement would close more of the long tail. `test/fuzzer/` hosts the existing evmone fuzzer; wiring it through the GPU dispatcher is on the v0.24 roadmap.

11 CUDA Port: Methodology and Inherited Bugs

The CUDA port of the GPU EVM kernel arrived as a sequence of three commits on the `feat/cuda-evm-kernel` branch (`b5ce473`, `16db493`, `2e3022a`, all merged Feb 2026), bringing the CUDA kernel from a stub at v0.21.0 to opcode-complete at v0.22.0 (`lib/evm/gpu/cuda/evm_kernel.cu`, 1713 SLOC).

11.1 Methodology

The Metal kernel was the source of truth. The port proceeded in four passes:

1. *Type carry-over.* The `uint256`, `pair64`, and storage-pool descriptor types defined in `kernel/uint256_gpu.hp` are shared between Metal and CUDA; only the address-space qualifiers differ (`device` vs `__device__`). The CUDA dialect uses `__host__ __device__` where the Metal version uses plain `static inline`.
2. *Bit-by-bit opcode translation.* Each opcode handler from Metal's `switch` block was translated into the corresponding CUDA case, preserving stack semantics, gas accounting, and emit-on-error behaviour. The Metal-specific bits — `threadgroup_barrier`, `simdgroup_*` — were replaced with CUDA equivalents (`__syncthreads`, `__shfl_sync`).
3. *Host launcher parity.* The Apple host (`evm_kernel_host.mm`) and the CUDA host (`cuda/evm_kernel_host.cp` implement the same `HostTransaction` ABI and dispatch through the same `gpu_dispatch.hpp` interface. The dispatcher's backend probe selects whichever device is present at runtime.
4. *Cross-validation.* The fourth backend was wired into the parity corpus (§10) under the `EVM_CUDA` define. The first run flushed the bugs catalogued below.

11.2 Three Inherited Bugs

The translation surfaced three consensus-relevant bugs that were already latent in the Metal kernel but had not yet been caught. The CUDA port acted as a differential oracle: identical bytecode through two implementations exposed where each was wrong against the EVM spec, not just against each other.

Bug 1: `u256_mulmod` carry-loss in upper limbs. The original 256-bit modular-multiply followed the textbook schoolbook loop, accumulating a 4×4 partial-product matrix into an 8-limb scratch buffer:

```
gpu_u64 r8[8] = {0, ..., 0};
for (uint i = 0; i < 4; ++i) {
    gpu_u64 carry = 0;
    for (uint j = 0; j < 4; ++j) {
        pair64 p = mul_wide(a.w[i], b.w[j]);
        s = r8[i+j] + p.lo;
        c = (s < r8[i+j]) ? 1 : 0;
        s += carry;
        c += (s < carry) ? 1 : 0;
        r8[i+j] = s;
        carry = p.hi + c;
    }
    if (i + 4 < 8) r8[i+4] += carry;  // BUG: i=3 lost
}
```

The guard `if (i+4 < 8)` is false for the last outer iteration (`i = 3`), so the carry leaving `a.w[3] * b.w[3]` was silently dropped. Even when the guard fired ($i \leq 2$), the naked `r8[i+4] += carry` could itself wrap around, and the secondary carry was never propagated to `r8[i+5]`.

The algebraic shape of the fix is to propagate the outer carry through *all* higher limbs:

$$\forall k \geq i+4: \quad (r'_k, c') := r_k + c \text{ (with } c' = \mathbf{1}[r_k + c < r_k]), \quad r_k \leftarrow r'_k,\ c \leftarrow c'.$$

In code (`evm_kernel.metal`, lines 204–239):

```
uint k = i + 4;
while (carry != 0 && k < 8) {
    gpu_u64 s = r8[k] + carry;
    gpu_u64 c = (s < r8[k]) ? 1UL : 0UL;
    r8[k] = s;
    carry = c;
    ++k;
}
```

The fix is identical in CUDA. The bug fires whenever $\lfloor a \cdot b / 2^{448} \rfloor > 0$, i.e. whenever the high limb of the product is non-zero, which is every input pair where both operands exceed 2^{192}. Real-world impact: any contract that uses `MULMOD` with values near group order in `secp256k1` or `bn254` (i.e. every contract that does on-chain elliptic-curve work) would see incorrect results. The bug was flagged `HIGH` in the internal red-team review before the parity corpus reproduced it.

Bug 2: Unrecognized opcodes did not consume all gas. The EVM spec mandates that an unrecognized opcode — not just the `0xFE INVALID` canonical reserved opcode, but any byte that is not a defined instruction — consumes all remaining gas of the current frame. The pre-fix kernel fell through to a generic `ERR()` macro that emitted `Error` status with the gas counter at its current value, which is wrong: the consensus-correct behaviour is to set `gas_used = gas_start` (the entire gas budget).

The fix is the new `ERRA()` macro on Metal kernel line 1059 (`ERR-all-gas`) which emits `(status=Error, gas=gas_start, output={})`. The `0xFE` canonical `INVALID` arm at line 1035 was already correct; the fix routes every unhandled opcode to the same path.

Bug 3: `MSTORE/MLOAD` offset+32 overflow. `MSTORE` (and `MLOAD`) take a 256-bit memory offset on the stack. The kernel implementation extracted the low 64 bits and performed the bounds check as

$$\mathsf{ok} := (\mathit{offset}_{\mathrm{low64}} + 32) \leq \mathsf{MAX\_MEM}.$$

For offsets near $2^{64} - 32$ (which is well below the EVM's $2^{256} - 1$ stack-representable range, but above the kernel's tx memory cap), the addition wrapped around and the comparison spuriously succeeded, allowing memory writes past the per-tx memory ceiling and corrupting the next transaction's slab in `mem_pool`. The fix (`evm_kernel.metal` lines 426–434) explicitly checks the high three limbs of the 256-bit offset, then the low 64 bits, then the addition for wraparound:

```
static inline bool offset_in_bounds(uint256 v, ulong extra) {
    if (v.w[1] | v.w[2] | v.w[3]) return false;
    if (v.w[0] > MAX_MEMORY_PER_TX) return false;
    ulong sum = v.w[0] + extra;
    if (sum < v.w[0])              return false;
    if (sum > MAX_MEMORY_PER_TX)   return false;
    return true;
}
```

The same algebraic check ports verbatim to CUDA.

11.3 What the LLVM Coverage Tells Us

LLVM source-based coverage on the kernel CPU interpreter (`evm_interpreter.hpp`, the spec witness) reports:

- **Line coverage**: 90.87%
- **Function coverage**: 100%
- **Region coverage**: ∼85% (target met)

The 9.13% of uncovered lines fall into three categories:

1. Unreachable arms behind `static_assert` guards (compile-time excluded but counted by the instrumenter);
2. The `CALL`/`CREATE` branches that intentionally return `CallNotSupported` — the parity test exercises these under the `KernelCpuMissing` expectation, but the line immediately following the early return is dead by design;
3. A handful of out-of-gas branches in `u256_exp` and `u256_mulmod` that fire only for inputs the corpus does not yet construct (extremely large bases with extremely large exponents that hit the `GAS_EXP_BYTE` multiplier ceiling).

The **100% function coverage** is the load-bearing claim: every opcode handler the kernel implements is entered by at least one parity vector. Combined with four-way agreement on bit-exact gas/status/output, this gives us a closed system: a divergence between any backend and any other backend is detectable by an existing test, on every commit.

What coverage does *not* certify is the path through Metal's AIR or CUDA's PTX after compilation. Metal compiles to AIR, which LLVM coverage cannot instrument; nvcc emits PTX without coverage-mapped IR. Coverage on the device kernels is therefore *indirect*: every parity vector that executes through the GPU dispatcher reaches the corresponding opcode handler in the device shader, and a divergence in that path manifests as an assertion failure in the parity test, not as a coverage gap. The four-way parity is the certification, not the coverage number.

11.4 CI Path: AWS EKS GPU Karpenter

The CUDA arm cannot run on the Apple-only developer workstation, so it runs on AWS EKS through GitHub-hosted Actions Runner Controller. `lux-gpu-ci/helm/karpenter-nodepool.yaml` declares an `EC2NodeClass` that provisions `g4dn.xlarge` (NVIDIA T4) spot instances on demand, with fallback to `g5.xlarge` (NVIDIA A10G) when T4 capacity is unavailable, scaling to zero 30s after the runner pod terminates. The runner-scaleset configuration ships the NVIDIA device plugin DaemonSet via Amazon Linux 2023 AMIs.

The CI workflow (`.github/workflows/cuda.yml` in the `luxcpp/evm` repository) runs nightly and on every PR that touches `lib/evm/gpu/cuda/**`. The job:

1. Configures CMake with `-DEVMONE_CUDA=ON -DCMAKE_CUDA_ARCHITECTURES="80;90"`;
2. Builds `evm-cuda` (the device library) and `evm-gpu` (the dispatcher with `EVM_CUDA` defined);
3. Verifies the CUDA symbols are present in the static archive (`nm libevm-cuda.a`);
4. Runs `evm-bench-kernel` as a smoke check;
5. In a dependent job, builds the `cevm` Go bindings against the freshly-compiled CUDA libraries and runs the Go test suite under `CGO_ENABLED=1`.

A nightly H100 self-hosted runner (`labels: self-hosted, gpu, cuda, h100`) runs the same workflow on `sm_90` hardware to validate the warp-shuffle paths added at v0.22.0; the EKS T4 runners cover `sm_80` and `sm_86`.

11.5 Limitations

- *No CUDA Block-STM yet.* The CUDA path implements single-tx-per-warp execution; the multi-version memory and re-execution scheduler in `block_stm.cu` compiles and links but is not yet wired into the dispatcher's `CPU_Parallel` equivalent on CUDA. Workloads above the M1 Max crossover therefore take the CPU Block-STM path on Linux/CUDA hosts; this is correct but leaves performance on the table. Path: complete the `BlockStmGpu` CUDA host launcher (`block_stm_host.cpp`) and fold it under `Backend::GPU_CUDA_Parallel`. Tracked.
- *Warp divergence on small batches.* A single tx whose bytecode contains long branchless arithmetic (e.g. a Solidity library doing 256-bit multiply chains) keeps all 32 threads of the warp busy; a tx with heavy data-dependent branching wastes warp lanes. The parity corpus does not yet quantify this. Path: instrument the kernel with `__activemask()` sampling and add a divergence-cost column to the bench output. Open.

- *No pre-Ampere coverage.* The CMake configuration targets `sm_80` (Ampere) and `sm_90` (Hopper); pre-Ampere GPUs are excluded because the kernel uses Ampere-only intrinsics in the keccak inner loop. Note: AWS `g4dn.xlarge` carries the T4 (`sm_75`, Turing) which falls below this threshold — the EKS Karpenter pool above is configured to fall back to `g5.xlarge` (A10G, `sm_86`) when T4 instances are incompatible with the build. The nightly H100 self-hosted runner (`sm_90`) is the canonical production validation target.

12 Empirical Results

This section consolidates the v0.23.0 measurements that other sections reference into a single table organised by backend and workload. Hardware: Apple M1 Max (32-core Metal GPU, 10-core CPU, 64 GB unified memory), macOS 14.5, clang/llvm 17. CUDA figures are produced by the nightly H100 runner (§11.4); the EKS T4/A10G runners cover build correctness rather than headline performance.

12.1 What Each Number Measures

- *TPS (transactions per second)* — end-to-end throughput including dispatch overhead. Measured on real EVM bytecode, not microbenchmarks.
- *Mgas/s* — aggregate gas processed per second. The EVM spec attaches a fixed gas cost to every opcode, so this number is the throughput a block builder cares about.
- *M ops/s* — raw opcode dispatch rate from `evm-bench-kernel`. Computed by counting bytecode opcodes actually executed (not gas-weighted) per loop iteration. Used primarily to compare GPU-V1 vs CPU evmone on the same compute-bound workload.

12.2 Headline Table

12.3 Where the GPU Loses

The bytecode kernel does not yet beat CPU_Parallel on every microkernel batch size. The detailed sweep (`lib/evm/gpu/results/speedup-with-buffer-cache-2026-02-28.txt`) shows GPU/CPU_Parallel ratios of 0.01×–0.23× across sweep cells $N \in \{32, 128, 512, 2048\}$ and 19 opcode workloads. The pattern is consistent: at $N \leq 512$, dispatch overhead dominates (∼6 ms–12 ms of cold-buffer cost vs ∼0.1 ms–1.2 ms for the CPU side). The GPU wins on:

- Workloads where `ecrecover` or large-input `KECCAK256` dominate (i.e. real blocks containing many transactions, not microkernels);
- Compute-heavy bytecode at $N \geq 4096$ where the 6 ms dispatch amortises over many transactions;
- Pipelined workloads where consensus state-hashing and sig-verify share the same Metal device with EVM execution.

The 1.62× headline figure on `evm-bench-kernel` is representative: pure-compute ADD-loop bytecode at N=2048 with all ecrecover paths excluded, comparing GPU-V1 (one tx per thread) to CPU evmone single-threaded. The same workload through CPU-Parallel does better in absolute terms but is not the relevant comparison for an ecrecover-dominated production block.

Workload	Backend	Throughput	Notes	Source
Real EVM bytecode (50-iter ADD loop, 13 ops/iter)				
N=2048 txs	GPU_Metal V1	25,327 TPS	1.14 M EVM ops/s	`bench_kernel.cpp`
N=2048 txs	GPU_Metal V1	1.14 M ops/s	gas-match vs CPU	`bench_kernel.cpp`
N=2048 txs	CPU evmone	155 M ops/s	single thread	`bench_kernel.cpp`
N=2048 txs	GPU_Metal V1	252 M ops/s	1.62× vs CPU	`bench_kernel.cpp`
Block-STM ETH transfers (21000 gas/tx, 10K txs/block)				
10K transfers	CPU_Sequential	534 Ggas/s	evmone, 1 thread	`bench_block.cpp`
10K transfers	CPU_Parallel (4T)	—	4-worker Block-STM	`bench_block.cpp`
10K transfers	CPU_Parallel (10T)	—	saturated cores	`bench_block.cpp`
Cryptographic critical path				
47K ecrecover	CPU	1613 ms	evmone batch	`bench_keccak.cpp`
47K ecrecover	GPU_Metal	50 ms	32.3×	`bench_keccak.cpp`
47K keccak256	CPU	127 ms	evmone keccak	`bench_keccak.cpp`
47K keccak256	GPU_Metal	18 ms	7.1×	`bench_keccak.cpp`
Unified pipeline (EVM + state hash + sig-verify)				
1000 blocks	Serial Metal	20 Hz	one stage at a time	`test_unified_pipeline.cpp`
1000 blocks	Pipelined Metal	19 Hz	1.07× overlap	`test_unified_pipeline.cpp`
Parity & correctness				
133 vectors	4 backends	100% pass	gas/status/output bit-exact	`parity_test.cpp`
147 vectors	GPU_Metal	100% pass	per-opcode coverage	`test_opcodes.cpp`
19 groups	GPU dispatch	100% pass	0x01–0x11 precompiles	`precompile_test.cpp`
7 scenarios	EVMC bridge	100% pass	CALL/CREATE pre-resolve	`host_bridge_test.cpp`

Table 5: Measured GPU EVM performance and correctness. All performance numbers are minimum-of-three runs to filter Metal device-coalescing warmup; correctness numbers are exact pass counts on the named test binaries. Source columns reference paths under `lib/evm/gpu/` and `test/unittests/` in the `luxcpp/evm` repository at tag `v0.23.0`.

12.4 Open Headline Numbers (v0.24+)

- *H100 baseline.* The H100 self-hosted runner exists but the headline table above is M1 Max only; H100 ops/s and TPS for the same bench are pending the v0.24 publication of the `cuda-bench-results` artefact.
- *End-to-end real-block TPS.* The numbers above are microbenchmarks. A measurement that ingests a real Cancun-era block from mainnet (with mixed precompile + storage + CALL workload) and reports wall-clock through the full GPU dispatcher is the next publication target.
- *CUDA Block-STM speedup.* CPU_Parallel exists; GPU_CUDA runs single-tx-per-warp. Pipelining the multi-version memory scheduler onto CUDA gives the obvious next $\sim 4\times$ on Block-STM-bound workloads, but the integration is incomplete (§11 limitations).

13 PQZ Cross-Chain Parallelism

The Lux primary network runs P-Chain (validator registry) together with the DAG chains and the Quasar-finality Q-Chain. We collectively call the operational model **PQZ**: Primary validators,

Quasar certificates, and the ZK overlay. The question: does running all chains concurrently on the same validator set compromise per-chain safety?

The answer is *no*, by a straightforward independence argument formalised in `Consensus/PQZParallel.lean`. The key observations:

1. **Type-level chain scoping**: vertex types in Go are package-parameterised (`zkvm.Vertex`, `dexvm.DexVertex`, etc.); a cross-chain `Conflicts` call is not expressible.
2. **Quasar certificates are chain-local**: each antichain Quasar stamp binds a chain-ID and rank; stamps from different chains are independently verifiable.
3. **Validator signing is re-entrant**: a validator signing on k chains performs k independent BLS + ML-DSA + Pulsar operations whose cost scales linearly in k. No chain blocks another.
4. **Shared validator set, shared network**: the only cross-chain coupling is bandwidth; GPU batch verify linearises signature cost across all chains on the same hardware.

We measured this empirically. The 6 DAG-capable VMs (Z, D, A, O, R, S) are driven by 17 unit tests running concurrently in `go test -race`; no data races are reported. The `evmgpu/dag` package adds another 17 tests (`TestStorageKeySetDisjoint` through `TestVertexStore`) with GPU and CPU build-tag variants. All pass.

Throughput implication: the Lux network's aggregate throughput is bounded below by the slowest DAG chain and above by the sum. With GPU ecrecover + GPU keccak + GPU bitmap-intersect shared across all chains (single context per host), adding chains is strictly additive.

14 The v0.24.0 Correctness Release

The two preceding sections describe a kernel that runs Cancun bytecode end-to-end (§8) and a consensus pipeline that plugs that kernel in via `block.ChainVM` (§9). Tag `v0.24.0` of `luxcpp/evm` is the first release whose explicit charter is *tighten the seam*, not extend it. No new opcodes, no new precompiles, no new backends. Eight specialist branches converged on a single dispatcher contract, opt-in safety flags, and an audit-driven plug of the holes that the v0.23.0 four-way parity corpus could not see. This section records what landed, what did not, and why the gap matters.

14.1 Adversarial Review Summary

Every Lux release goes through a Red/Blue audit cycle: a Red specialist walks the diff against the EVM spec hunting consensus-relevant defects, Blue triages and assigns each finding to a specialist branch, and the release is cut only when every fix branch passes the four-way parity corpus from §10. For `v0.24.0` Red filed **22 HIGH-severity consensus findings** against the v0.23.0 head. "HIGH" means the defect can produce a bit-different `(gas_used, status, output)` tuple from the EVM spec on at least one constructed input.

Table 6 groups the findings by the layer they hit and the resolution status at `v0.24.0`. The honest accounting matters: of the 22, eight closed in this release, six are partial (the fix is on a specialist branch but waiting for parity-corpus extension), and eight are explicitly deferred to v0.25.0 with named owners.

The point is not the count. Adversarial review on every version produces *a public list of defects with named owners and verifiable resolution commits*. Eight HIGH findings deferring to v0.25.0 is the truthful state of an EVM that targets consensus-grade behaviour. A four-way parity corpus is necessary but not sufficient: the Red review is the missing half of the diff oracle.

Finding (Red ID)	Layer	Status at v0.24.0
Stack 32 → 1024 entries (R1)	Metal kernel	DONE (`8487d81`)
Mulmod 128-bit carry (R2)	CUDA kernel	DONE (`a70ecf3`)
Default-case ERRA semantics (R3)	Metal kernel	DONE (`2ab4b4d`)
Offset-in-bounds checks (R4–R5)	Metal kernel	DONE (`2642e5d`)
Refund counter signed int64 (R6)	Metal kernel	DONE (`c8c8845`)
JUMPDEST O(1) bitmap (R7)	CUDA kernel	DONE (`be6714c`)
EIP-3529 cap refund / 5 (R9)	Metal kernel	DONE (`bc72505`)
SSTORE/TSTORE overflow (R10)	Metal host	DONE (`5bdc7d4`)
CallNotSupported leaks to caller (R11)	Dispatcher	DONE (`0a5b8b7`)
Gas-estimation shortcut unguarded (R12)	Dispatcher	DONE (`b6ccd82`)
Precompile positive-tests gap (R13)	Test corpus	DONE (`6f33dfd`)
ABI v3 init guard (R14)	cevm Go	DONE (`4d652b2`)
Health() coverage gaps (R15)	cevm Go	DONE (`cf8ecb7`)
EIP-2929 access-list pre-warm (R16)	Dispatcher	PARTIAL (flag only, no kernel wire)
Cap-overflow fallback (R17)	Dispatcher	PARTIAL (Metal only)
Account-state routing (R18)	Host bridge	PARTIAL (CALL bridge only)
Unified pipeline non-Apple (R19)	Host launchers	PARTIAL (compile-only on Linux)
V2 Metal kernel implementation (R20)	Metal kernel	DEFERRED to v0.25
Async dispatch API (R21)	Metal host	DEFERRED (`fc28203` on branch)
GpuHostBridge real GPU CALL (R22)	Host bridge	DEFERRED to v0.25

Table 6: Red audit findings on `luxcpp/evm` for the v0.24.0 release window. Commit hashes refer to fix-landing commits on `luxcpp/evm` (i.e. the SHA where the patch entered a specialist branch; merges into `main` use a separate hash). "PARTIAL" means the fix lands on the dispatcher or kernel but the parity-corpus extension that would catch a regression is still in review. "DEFERRED" is an explicit owner-and-deadline assignment.

14.2 Backend Routing as a State Machine

Before `v0.24.0` the dispatcher had a tangle of `if backend == Metal && has_code && has_host ...` branches in a single `execute_block` function. The Red audit found two routing bugs (`CallNotSupported` leaking to caller; gas-estimation shortcut firing without an opt-in) that were trivially detectable once the routing was drawn as a table. Commit `0a5b8b7` on `feat/specialist-dispatch` refactors `execute_block` into three per-backend functions (`run_cpu`, `run_metal`, `run_cuda`) that each consult the same documented routing tree. The tree has 16 cells: 4 backends × 2 (has-code) × 2 (has-host).

CallNotSupported as an internal signal. `TxStatus::CallNotSupported` (status 5) is the kernel's way of saying "I cannot resolve this `CALL` — run me on CPU." Before v0.24.0 it leaked to the caller as if it were a consensus-relevant `TxStatus`. Commit `0a5b8b7` adds `apply_call_not_supported_fallback`, an unconditional post-kernel pass that for each `CallNotSupported` tx either re-runs through `execute_sequential_evmone` (when a host was supplied; status normalised back to `Stop/Return/Revert/OutOfGa` or rewrites to `Error` with a populated `error_message`. The dispatcher also increments `BlockResult::gpu_fallbac` so callers see the GPU rejection rate without parsing per-tx status arrays. The fallback is symmetric across all four backends; the CPU bytecode path goes through the same filter so a parity test tagged `Agree` sees identical results across all four routes.

Backend	has-code	has-host	Route
CPU_Sequential	no	no	gas-estimation gate or Error
CPU_Sequential	no	yes	evmone (sequential, real host)
CPU_Sequential	yes	no	kernel::execute_cpu (parity reference)
CPU_Sequential	yes	yes	evmone (sequential, real host)
CPU_Parallel	no	no	gas-estimation gate or Error
CPU_Parallel	no	yes	evmone Block-STM (real host)
CPU_Parallel	yes	no	kernel::execute_cpu per-thread
CPU_Parallel	yes	yes	evmone Block-STM (real host)
GPU_Metal	no	no	BlockStmGpu scheduler-only or Error
GPU_Metal	no	yes	evmone Block-STM (real host)
GPU_Metal	yes	no	EvmKernelHost + CNS fallback
GPU_Metal	yes	yes	evmone Block-STM (real host)
GPU_CUDA	no	no	BlockStmGpu scheduler-only or Error
GPU_CUDA	no	yes	evmone Block-STM (real host)
GPU_CUDA	yes	no	cuda::EvmKernel + CNS fallback
GPU_CUDA	yes	yes	evmone Block-STM (real host)

Table 7: Backend routing matrix at v0.24.0. The 16 cells each map to exactly one execution path. The has-host/has-code axes are *both* runtime inputs to execute_block(config, txs, host); the third axis (config.gas_estimation_mode, config.fast_value_transfer ...) is opt-in and changes the no-code/no-host cells from Error to a defined approximation. Source: lib/evm/gpu/gpu_dispatch.cpp on feat/specialist-dispatch (commit 0a5b8b7). The 15-test evm-dispatch-test target (test/unittests/dispatch_test.cpp, 487 SLOC, commit 0223eeb) covers every cell at least once.

14.3 Opt-in Optimisation Flags

Real consumers of cevm need three things the spec EVM does not provide: a fast gas-estimation mode for fee oracles, a low-overhead value-transfer fast path for benchmarks, and an EIP-2929 access-list pre-warm that lifts the cold-storage tax for known-warm slots. None of these is consensus-safe in general. Commit b6ccd82 adds five flags to Config, all default-off:

```
struct Config {
    Backend  backend                          = Backend::CPU_Sequential;
    uint32_t num_threads                      = 0;
    uint32_t gpu_device                       = 0;
    bool     enable_state_trie_gpu            = false;
    bool     enable_precompile_gpu            = false;

    // v0.24.0 opt-in flags ------------------------------------------
    bool     gas_estimation_mode              = false;
    bool     fast_value_transfer              = false;
    bool     fast_value_transfer_acknowledged = false;
    bool     skip_parity_check                = false;

    std::vector<uint8_t> warm_addresses;       // 20-byte each
    std::vector<uint8_t> warm_storage_keys;    // 52-byte each (addr || slot)
};
```

The safety contract for each flag, verified by test/unittests/dispatch_test.cpp:

`gas_estimation_mode`. Permits the no-host/no-code shortcut that returns `gas_used = gas_limit`. Useful for fee oracles. Without the flag the dispatcher returns `Error`; a caller cannot mistakenly treat a placeholder gas number as real execution. Test: `TEST(Dispatch, GasEstimationGate_DefaultOff)`.

`fast_value_transfer` + `fast_value_transfer_acknowledged`. Two-step lock. The first flag alone produces `Error` with `error_message = "fast_value_transfer requires explicit acknowledgement"`. Both required to enable. The dispatcher logs to stderr on every block when set. **Never consensus-safe** — this is benchmark-only, for measuring kernel dispatch cost without an evmone interpreter loop. Test: `TEST(Dispatch, FastValueTransferAckRequired)`.

`warm_addresses`, `warm_storage_keys`. Pre-warmed access list per EIP-2929. Empty = everything cold. Flat 20-byte / 52-byte vector layout avoids a per-block std::map allocation. Wired through the dispatcher contract today; the kernel-side surcharge skip lands on `feat/blue-hardening` but is not yet in the parity corpus (R16 PARTIAL).

Semantics summary. With every flag default-false, v0.24.0 preserves bit-exact EVM consensus across every cell of Table 7. With unsafe flags set, the dispatcher logs to stderr on every block and the result is documented as approximate. The right place is benchmark and fee-oracle code; the wrong place is mainnet validators.

14.4 Empirical Performance Ceiling

The fair speedup sweep on M1 Max after the buffer-cache fix (`lib/evm/gpu/PERF.md`) shows the GPU *losing* on typical block sizes. Reproduced verbatim from `PERF.md` at `v0.24.0`:

Path	N	**Speedup**	**Source**
Direct kernel API (no dispatcher overhead)			
`bench_kernel.cpp` 11M opcodes	10K × 100	1.84×	GPU 339 M ops/s
Dispatcher API (`cevm.ExecuteBlock` via `gpu_dispatch.cpp`)			
N=4K compute-loop bytecode	4096	0.31×	CPU 8.5ms / GPU 27.4ms
N=16K compute-loop bytecode	16384	0.46×	CPU 30.7ms / GPU 66.6ms
N=32K compute-loop bytecode	32768	0.53×	CPU 60.0ms / GPU 114.1ms
N=64K compute-loop bytecode	65536	0.48×	CPU 119.7ms / GPU 249.3ms

Table 8: M1 Max speedup vs CPU_Parallel after the MTLBuffer cache landed (`48a0495` on `feat/speedup-bench`, 2026-02-28). The dispatcher curve plateaus at 0.53× around $N = 32$K and *regresses* at $N = 64$K — L1-cache pressure as buffer count grows. The direct-kernel path wins because it ships an 11M-opcode workload in a single dispatch; the dispatcher path ships ∼2K–600K opcodes per call, an order of magnitude below the amortisation threshold. Source: `lib/evm/gpu/PERF.md`.

Crossover. The break-even calculation in §9.6 placed the GPU-wins-on-bytecode threshold at ∼1.2M opcodes per dispatch. The dispatcher's typical workload is two orders of magnitude below that. Only the `bench_kernel` path crosses, by shipping 10K txs × 100-iter loops in one launch. The honest reading: *at the workloads a Lux validator actually emits per `Verify` call, the GPU loses on bytecode.* The crypto-dominated path (§8.5) still wins because ecrecover/keccak per-op cost on CPU is higher.

Root causes (PERF.md). Three:

1. *V2 kernel declared in host but not defined in Metal source.* `evm_kernel_host.mm` looks for `evm_execute_v2`, a 32-thread/tx SIMD-cooperative kernel that would saturate M1 Max's 4096 simultaneous lanes. The Metal source defines only `evm_execute` (1 thread/tx); `has_v2()` returns false; every call lands on V1.

2. *Per-tx scratch buffer over-provisioned at 64 KB.* The kernel reserves 64 KB per tx regardless of need; a 50-iter ADD-loop tx uses 64 bytes. memset dominates at small batches. Commit `3003b45` on `feat/v0.24-correctness` adds static-analysis per-batch sizing (R17 cap-overflow fallback).

3. *Synchronous `[commandBuffer waitUntilCompleted]` blocks the dispatcher.* The async API lands on `feat/specialist-metal-host` (`fc28203`) but is not yet promoted into the dispatcher.

`v0.24.0` ships as a correctness release; V2 kernel, async dispatch, indirect command buffers, and persistent state are the four priorities for v0.25.0. The bytecode kernel is correct; the performance shape is what remains.

14.5 Four-Way Parity Tightening

The four-way parity corpus from §10 is the only mechanism that catches a backend silently producing a wrong gas number on a vector the corpus already touches. `v0.24.0` extends the corpus along three axes:

Routing matrix coverage. `test/unittests/dispatch_test.cpp` (commit `0223eeb`, 487 SLOC, 15 GoogleTest cases) walks every cell of Table 7 at least once and checks the `(status, gas_used, error_message, gpu_fallback_count)` tuple against an expected outcome:

- `TEST(Dispatch, CallNotSupportedFallback_WithHost)` — a `CALL` bytecode under each backend, with an `evmc::Host*` supplied, must surface as `Stop` not `CallNotSupported`.
- `TEST(Dispatch, CallNotSupportedFallback_NoHost)` — the same bytecode with no host must surface as `Error` with a populated `error_message`, never as `CallNotSupported`.
- `TEST(Dispatch, FastValueTransferAckRequired)` — setting `fast_value_transfer` alone must produce `Error`; setting both flags must succeed (with a stderr log expected).
- `TEST(Dispatch, GasEstimationGate_DefaultOff)` — no-host/no-code path returns `Error` unless `gas_estimation_mode` is set.
- `TEST(Dispatch, BackendRouting_*)` (12 cases, one per {backend, has-code, has-host} cell) — each backend takes the documented route and produces the documented status.

Precompile positive-test gap closure. Commit `6f33dfd` on `feat/specialist-precompile-tests` adds 11 positive-functional tests for precompiles `0x08` (BN256_PAIRING) through `0x11` (BLS12_MAP_FP2_TO_G2) filling the gap surfaced by Red finding R13. Each test pulls a vector from the canonical EIP source:

- `0x08 BN256_PAIRING`: empty input + EIP-197 "jeff1" (two cancelling pairs).

- `0x09 BLAKE2F`: EIP-152 test vector 4 (BLAKE2b("abc"), rounds=12).
- `0x0a POINT_EVALUATION`: EIP-4844 zero-polynomial KZG proof.
- `0x0b`–`0x11`: EIP-2537 BLS12-381 test corpus (G1ADD, G1MSM, G2ADD, G2MSM, PAIRING_CHECK, MAP_FP_TO_G1, MAP_FP2_TO_G2).

The precompile test count moved from 19 to 30. Every vector is byte-exact against the published EIP corpus; gas consumption is checked against the spec formula at the dispatcher boundary. (The pre-existing tests for `0x08`–`0x11` only covered failure paths — "invalid input size returns OutOfGas" — which gave zero confidence that the correct-input path produced correct output.)

Outstanding gap: `KernelCpuMissing` carve-outs. The spec witness (`evm_interpreter.hpp` from §10) still does not implement 26 opcodes per Red's count, each tagged `Expectation::KernelCpuMissing` in the parity corpus. The `feat/specialist-cpu-interpreter` branch is closing them, rate- limited by reviewer bandwidth. Until the carve-outs are zero, a CPU-only v0.24.0 deployment fails on those opcodes (the dispatcher returns `Error`, not a wrong gas number, but the deployment shape "CPU only, no GPU available" is degraded).

14.6 Roadmap to v0.25.0

The five v0.25.0 priorities, in landing order:

1. *V2 Metal kernel.* Define `evm_execute_v2` in `lib/evm/gpu/kernel/evm_kernel.metal`: 32 threads/tx cooperative, warp-shuffle stack ops. PERF.md estimate: 3–5× at $N = 2$K. Owner: `feat/specialist-evm-kernel-metal`.
2. *Async dispatch in production.* Promote `execute_async` (`fc28203`) into `gpu_dispatch.cpp`. Risk: the integration in §9.4 assumes a synchronous `Verify`; async dispatch needs a callback shape on `cevmBlock`.
3. *Full CPU interpreter parity.* Close the 26 missing opcodes on `feat/specialist-cpu-interpreter` so every `Expectation::KernelCpuMissing` promotes to `Agree`.
4. *EIP-2929 access-list implementation.* Wire `warm_addresses`/`warm_storage_keys` into the kernel's per-opcode gas decrement; today the flags exist on the dispatcher contract but the kernel charges full cold cost. Owner: `feat/blue-hardening`.
5. *GpuHostBridge real GPU CALL.* Currently a 4-deep static-CALL chain falls through to evmone. Real GPU CALL needs the kernel to pop a frame from a dictionary entry, push it on a per-thread call stack, and continue with new code/data pointers. Owner: `feat/specialist-evmc-bridge`.

These five close the v0.24.0 "DEFERRED" lines of Table 6 and clear the bytecode crossover threshold. Item 1 alone upper-bounds the ceiling in §14.4; items 2–5 buy the rest.

Net. v0.24.0 is the release where the seam between kernel and consumer became contractually clean: documented routing matrix, opt-in unsafe flags with explicit acknowledgements, fallback paths verified end-to-end by 15 dispatcher tests + 30 precompile tests + the 133-vector parity corpus. The performance promise is unchanged from v0.23.0; the gap to the target is documented precisely. v0.25.0 is where it closes.

15 The v0.26.0 Benchmark Re-run and Corrections

The v0.24 release (§14) closed eight HIGH-severity Red findings and introduced the dispatcher contract that v0.25 hardened with signed refunds, EIP-3529 capping, and storage-overflow detection. v0.26 is in flight at the time of writing. This section reports a fresh M1 Max benchmark run against the current `luxcpp/evm` `main` head (`a4b26457e921`) to (a) replace the v0.24 numbers in §14.4 that did not record their statistical methodology and (b) surface a Cancun-completeness consensus bug uncovered by the new MCOPY workload.

15.1 Method

- **Hardware.** Apple M1 Max, 8 performance + 2 efficiency cores, 32-core Metal GPU, 64 GB unified memory, macOS `25.4.0` (Darwin `25.4.0`), system idle (no other active workloads).
- **Toolchain.** Go 1.26.2, clang 17, Metal Shading Language 3.
- **Backends.** CPU-Sequential, CPU-Parallel (Block-STM), GPU-Metal. CUDA is omitted; the M1 Max has no CUDA device.
- **Driver.** `luxfi/benchmarks` branch `feat/v0.26-bench-results`, command `cmd/v026-bench`. Produces a single self-contained JSON report with hardware metadata, every cell's wall clock, and the canonical bytecode hex for each workload.
- **Sample size.** Five timed runs per (workload, N, backend) cell after a 16-tx warmup. We report the *median* wall-clock, not the minimum. The minimum is reported separately for the `evm-bench-kernel` re-run only, so that the v0.24 1.84× figure is comparable on its original methodology.
- **Sweep.** $N \in \{128, 1024, 8192\}$.
- **Workloads.** Five canonical bytecodes, listed below. Each is deterministic; the JSON report includes the hex.

Workload bytecodes

The five workloads cover the four EVM cost classes (compute, hashing, storage, memory) plus a realistic mixed workload (ERC-20 transfer):

1. `Compute50` — 50 iterations of `(PUSH1 7, PUSH1 5, ADD, PUSH1 3, ADD, POP)` followed by `RETURN`. 300 stack ops per tx, 856 gas per tx, no memory, no storage, no calldata. Measures pure opcode dispatch.
2. `Keccak10` — 10 iterations of `(PUSH1 32, PUSH1 0, KECCAK256, PUSH1 0, MSTORE)` followed by `RETURN`. 50 ops per tx. Exercises the `KECCAK256` primitive over a fixed 32-byte memory window.
3. `ERC20Canonical` — the canonical ERC-20 `transfer` body: `SLOAD, SUB, SSTORE, SLOAD, ADD, SSTORE, LOG3, RETURN`. 18 dispatched ops, 45 748 gas per tx, realistic mixed compute + storage + log workload.
4. `Storage10` — 10 iterations of `(PUSH val, PUSH slot, SSTORE, PUSH slot, SLOAD, POP)` against ten distinct slots. EIP-2929 cold-SSTORE pricing produces 221 116 gas per tx (10 × 22 100 plus dispatch).

5. `MCopy32K` — one priming `MSTORE` followed by 32 `MCOPY` ops of 1024 bytes each, total 32 KB moved. Exercises the EIP-5656 opcode landed in v0.25.

15.2 Per-Workload Results (Median Wall Clock)

Tables 9–13 record the median of five runs per cell. The complete JSON dataset including standard deviation, min, max, and per-tx gas is at `benchmarks/results/v0.26.0-bench-2026-03-05.json`.

N	CPU-Seq (ms)	CPU-Par (ms)	GPU-Metal (ms)	GPU/Par
128	0.42	0.86	161.9	0.01×
1 024	3.07	3.30	346.6	0.01×
8 192	27.12	20.54	189.9	0.11×

Table 9: `Compute50` (50-iter ADD loop). At $N = 8192$ the CPU-Par processes 398 754 TPS and the GPU processes 43 130 TPS through the dispatcher. The GPU loses by 9.2×.

N	CPU-Seq (ms)	CPU-Par (ms)	GPU-Metal (ms)	GPU/Par
128	0.24	3.10	141.2	0.02×
1 024	2.02	2.06	246.1	0.01×
8 192	17.96	15.27	331.0	0.05×

Table 10: `Keccak10`. Note: CPU reports `gas_used = 6` per tx, GPU reports `gas_used = 489` per tx. The kernel CPU interpreter (`lib/evm/gpu/kernel/evm_interpreter.hpp`) does not implement `KECCAK256` (opcode `0x20`); it falls through `InvalidOpcode`, and the dispatcher silently coerces the result to `TxStop`. The CPU TPS row is therefore measuring an empty execution and overstates real throughput. See §15.4.

N	CPU-Seq (ms)	CPU-Par (ms)	GPU-Metal (ms)	GPU/Par
128	0.25	0.55	133.2	0.00×
1 024	1.94	1.89	257.0	0.01×
8 192	20.07	15.38	301.9	0.05×

Table 11: `ERC20Canonical`. CPU and GPU agree on gas_used = 45 748 per tx; this workload exercises the storage path correctly on both backends.

15.3 Where the GPU Wins, Where it Loses

Across every cell of the dispatcher path (`cevm.ExecuteBlock` / `gpu_execute_block`) at $N \in \{128, 1024, 8192\}$ the GPU loses to CPU-Parallel by a factor between 0.00× and 0.11×. The reason is structural: each dispatcher call carries an approximately 120–540 ms cold-launch cost (buffer setup, shader resource binding, command-queue submission, result readback). At $N = 8\,192$ the CPU completes a 21-million-op `Storage10` batch in 18 ms; the GPU spends 341 ms on the launch alone. The cold cost does not amortise within a single dispatcher call because the dispatcher releases the Metal context between calls.

The GPU wins on the *kernel-direct path*, which bypasses the dispatcher and submits one launch over a many-tx batch. The canonical `evm-bench-kernel` test (10 K txs × 100-iter compute, 11.04 M opcodes) runs through this path. We re-measured it seven times under identical thermal conditions:

N	CPU-Seq (ms)	CPU-Par (ms)	GPU-Metal (ms)	GPU/Par
128	0.59	0.40	123.8	0.00×
1 024	2.34	1.76	277.3	0.01×
8 192	21.91	18.47	341.2	0.05×

Table 12: `Storage10`. CPU and GPU agree on gas_used = 221 116 per tx, matching EIP-2929 cold-SSTORE pricing.

N	CPU-Seq (ms)	CPU-Par (ms)	GPU-Metal (ms)	GPU/Par
128	0.23	0.33	170.6	0.00×
1 024	1.95	9.23	299.2	0.03×
8 192	16.53	15.59	543.9	0.03×

Table 13: `MCopy32K`. *The CPU reports gas_used* = 21 *per tx; the GPU reports gas_used* = 8 817 *per tx.* The CPU is producing wrong results — see §15.4. Even the *wrong* CPU result executes in < 1 ms, while the correct GPU result takes 540 ms; through the dispatcher path this workload is 30× slower on GPU.

The GPU wins on the kernel-direct path. The win is real but smaller than v0.24 reported when measured with median statistics, and larger when measured with the same min-of-runs methodology. See §15.5.

15.4 Consensus-Affecting Bug: Missing Cancun Opcodes on CPU Path

The `Keccak10` and `MCopy32K` workloads above exposed a class of bugs that the four-way parity corpus (§10) does not catch: the kernel CPU interpreter at `lib/evm/gpu/kernel/evm_interpreter.hpp` is missing case arms for opcodes that *do* appear in the GPU kernels.

The interpreter implements opcodes $\{0x00, \ldots, 0x4f\}$ ENV / COMPUTE, $\{0x50, 0x51, 0x52, 0x53, 0x59\}$ POP / MEMORY, $\{0x54, 0x55\}$ STORAGE, $\{0x56, 0x57, 0x58, 0x5a, 0x5b\}$ CONTROL, `0x5f` PUSH0, $\{0x60, \ldots, 0x9f\}$ PUSH/DUP/SWAP, $\{0xa0, \ldots, 0xa4\}$ LOG, and routes $\{0xf0, 0xf1, 0xf2, 0xf4, 0xf5, 0xfa, 0x$ to `CallNotSupported` for the host bridge. Every other opcode falls through to a final `InvalidOpcode` arm.

The interpreter has no case for `0x20` KECCAK256, no case for `0x5c` TLOAD, no case for `0x5d` TSTORE, and no case for `0x5e` MCOPY. The Metal kernel (`lib/evm/gpu/kernel/evm_kernel.meta` and the CUDA kernel (`lib/evm/gpu/cuda/evm_kernel.cu:1526`) both implement MCOPY correctly. The dispatcher then converts the `InvalidOpcode` return from the kernel CPU path into `TxStop` when the dispatcher's status mapping table treats it as a clean halt. The result is that a Cancun-era block containing any of these four opcodes will produce a different `(gas_used, status, output)` tuple between the CPU and GPU backends.

Severity. Consensus-affecting. A validator running the GPU backend and a validator running the CPU backend will disagree on the gas accounting of any block with MCOPY, TLOAD, TSTORE, or KECCAK256 in non-trivial bytecode. Mainnet has been on Cancun since 2024-03-13. Every modern ERC-20 / ERC-721 / Uniswap-style contract emits KECCAK256 in its `transfer`; many emit MCOPY since solc 0.8.25.

Mitigation in flight. The v0.26 milestone has named owners to land KECCAK256, MCOPY, TLOAD, TSTORE in `kernel::execute_cpu`. The four-way parity corpus (§10) needs an exten-

Metric	CPU	GPU V1	Speedup
Best wall (min of 5)	70.0 ms	20.0 ms	3.50×
M ops/s (best-of)	158 M/s	552 M/s	—
Median wall (across 7 outer runs)	71 ms	55 ms	1.29×
M ops/s (median)	155 M/s	200 M/s	—
Std-dev / mean (GPU)	—	95%	—

Table 14: Re-run of `evm-bench-kernel` on the v0.26 main head. "Best of" uses the same min-of-runs methodology as the v0.24 paper table. "Median" is the proper statistical reading.

sion covering at least {KECCAK256 with non-zero memory expansion, MCOPY with overlapping ranges, TLOAD/TSTORE within and across nested call frames} before v0.26 can be tagged. Until that lands, *the parity claim from §10 is valid only for non-Cancun bytecode.*

15.5 Corrections to Prior Numbers

Three claims in earlier sections of this paper need correction in light of the v0.26 measurements above.

Claim 1 (§14.4, v0.24): "`bench_kernel.cpp` 11M opcodes, 10K × 100, 1.84× speedup, GPU 339 M ops/s". The v0.24 paper recorded a single best-of-runs number with no methodology disclosure (no N, no whether it was median or min, no trial count). Today's seven-trial re-run produces 3.50× best-of-runs at 552 M ops/s, and 1.29× median across 7 trials at 200 M ops/s. The GPU is in fact *faster* than v0.24 reported on its own min-of-runs methodology, but the GPU has $\sim 95\%$ σ/μ variance trial-to-trial, so a single-number summary is misleading. The corrected statement: "GPU wins on `evm-bench-kernel` 10K × 100 by 3.50× best-of and 1.29× median; the spread reflects Metal scheduling jitter under thermal load and is not a fundamental architectural variance."

Claim 2 (§12, v0.23): "GPU_Metal V1 252 M ops/s, 1.62× vs CPU evmone, $N = 2048$ `evm-bench-kernel`". This number compared GPU_Metal against the legacy evmone interpreter, not the kernel CPU interpreter. The v0.26 dispatcher routes the CPU parity path through `kernel::execute_cpu`, which is faster than evmone at this microbenchmark and narrows the GPU's apparent lead. The 1.62× figure cannot be reproduced as written because the baseline interpreter has changed. The corrected statement: "On `evm-bench-kernel` with the v0.26 baseline (kernel CPU interpreter, not legacy evmone), GPU/CPU is 3.50× best-of and 1.29× median at 11M opcodes; the v0.23 1.62× figure is unreproducible because its baseline interpreter has been replaced."

Claim 3 (§12, v0.23): "Pipelined Metal 1.07× overlap" and the table row labelled "Unified pipeline (EVM + state hash + sig-verify)". The pipeline test in v0.23 conflated three separate stages and reported a single 1.07× overlap. We did not re-run this workload for v0.26 because the unified pipeline rebuilt against v0.26 ABI v4 still has open holes from the v0.24 deferred list. The corrected statement: "Pipelined throughput is not yet quotable on v0.26; the rewrite tracking the v0.24 deferred unified-pipeline items is the gating dependency."

15.6 Architectural Conclusions and the Path to v0.27

Three structural observations follow from the data above.

The dispatcher path is a parity / correctness fallback, not a performance path on Apple silicon. The fixed $\sim 120{-}540$ ms cold-launch cost dominates any amortisation possible at $N \leq 8\,192$, which covers every realistic Lux block size today. The kernel-direct path is where the GPU wins. v0.27 should (a) document the kernel-direct API as the recommended block- builder integration, and (b) either eliminate the cold cost via a persistent Metal command queue retained across blocks, or gate the GPU dispatcher to $N \geq\approx 32\,768$ with deterministic CPU fall-through below that threshold.

The kernel CPU interpreter is not Cancun-complete. Until v0.26 lands KECCAK256, MCOPY, TLOAD, and TSTORE in `kernel::execute_cpu`, the four-way parity claim (§10) is valid only for pre-Cancun bytecode. Mainnet has been on Cancun since 2024-03-13. The fix is local to one file ($\sim$ 200 lines of opcode handling); the harder part is extending the parity corpus with adversarial Cancun cases before re-asserting the claim.

Variance reporting is a tooling problem. The 95% σ/μ on `evm-bench-kernel` GPU runs is large enough that any single-number summary mis-states the architecture. The fix is harness-level: report median + IQR + $p99$, warm for at least 5 runs, pin to a known QoS class to remove macOS scheduler jitter, record the thermal state of the device. The min-of-runs convention inherited from `evm-bench-kernel` is acceptable as a ceiling but should never be the headline number in a release artefact.

What v0.27 should ship.

1. `kernel::execute_cpu` with KECCAK256, MCOPY, TLOAD, TSTORE. Re-issue the four-way parity corpus extension.
2. Persistent Metal command queue in the dispatcher path; target is a 10× reduction in cold-launch cost. Below that, the dispatcher remains a parity path only.
3. Bench harness with median + IQR + $p99$ as the canonical output. Min-of-runs as a separately labelled ceiling number.
4. A real end-to-end real-block measurement on a captured Cancun-era mainnet block (mixed precompile + storage + CALL), once the unified pipeline rewrite lands.

The bottom line is unchanged from v0.24: the GPU is the right place for keccak, ecrecover, and large-batch kernel-direct execution; the CPU remains the right place for the dispatcher path on M1 Max and for the parity-and-correctness fallback. v0.26 closes the Cancun hole on the CPU side; v0.27 closes the cold-cost hole on the GPU side.

16 Conclusion

We have shown that the Lux multi-chain admits a uniform DAG consensus model in which the execution engine's conflict graph *is* the consensus DAG, yielding a single implementation pattern

that covers EVM, UTXO, ZK-UTXO, DEX, Oracle, Relay, AI, and Service-Node workloads. Safety, serialisability, and double-spend prevention are proved in Lean 4 and model-checked in TLA$^+$.

The GPU EVM stack delivered through tags `v0.22.0`, `v0.23.0`, and `v0.24.0` of the `luxcpp/evm` repository extends the critical-path argument: ecrecover and keccak retain their original 32.3× and 7.1× speedups, the Metal kernel is opcode-complete on Cancun, the CUDA port reproduces the Metal kernel under a four-way parity test (133 hand-crafted bytecode vectors, four independent backends, bit-exact agreement on gas/status/output), and 100% function coverage on the kernel CPU spec witness ensures every implemented opcode is exercised on every CI run. Three consensus-relevant bugs (mulmod carry-loss, unrecognized-opcode gas accounting, MSTORE offset overflow) were surfaced by the differential-oracle property of cross-implementation parity and fixed in both kernels. The v0.24.0 correctness release (§14) contractualises the dispatcher–consumer seam, closes 8 of 22 Red HIGH findings outright, and documents the precise performance ceiling — with five named priorities for v0.25.0 to clear the bytecode crossover threshold.

The formal development is open-source at `lux/formal`; the benchmarks are at `lux/evm-bench` and `lux/benchmarks`; the Go implementation is at `lux/evmgpu` and `lux/chains`; the GPU EVM kernel and parity test driver are at `luxcpp/evm` under tag `v0.24.0`.

References

[1] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, R. Zhou. *Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing*. PPoPP 2023.

[2] K. Babel, A. Chursin, G. Danezis, L. Kokoris-Kogias, A. Sonnino. *Mysticeti: Reaching the Limits of Latency with Uncertified DAGs*. arXiv:2310.14821.

[3] J. Tang et al. *Monad: Decoupling Execution and Consensus for Parallel EVM*. Monad Labs whitepaper, 2024.

[4] V. Yin, M. Ho. *pevm: Parallel EVM*. Risechain whitepaper, 2024.

[5] Paradigm. *Reth: A Rust Ethereum Execution Layer Client*. 2024.

[6] Ipsilon. *evmone: Fast Ethereum Virtual Machine Implementation*.

[7] G. Danezis, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman. *Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus*. EuroSys 2022.

Reproducibility

Source code. `github.com/luxfi/consensus` (commit `TBD`); EVM at `github.com/luxfi/evm`; node at `github.com/luxfi/node`.
Build. `go build ./...` and `go test ./...` at the corresponding release tag; CUDA port requires NVCC $\geq$ 12.
Hardware tested. TBD (x86_64 Linux; NVIDIA GPU for the CUDA execution path).
Standards referenced. EIPs cited inline (London/Cancun/Prague feature gates); FIPS 204 (ML-DSA) for the post-quantum cross-chain lane.
Contact. security@lux.network.