Lux Lightspeed DEX: High-Frequency Trading at the Speed of Light

Zach Kelling

Lux Industries

Abstract

We present **Lux Lightspeed DEX**, a decentralized exchange whose matching engine *is* a chain. The DEX is the **D-Chain**: a single-operator high-frequency-trading consensus chain with its own consensus, `zapdb` persistence, and an execution-root state commitment, and the only place orders match or DEX state lives. A stateless **atomic ZAP proxy** (`chains/dexvm`) on the public Lux network relays orders to the D-Chain and moves value atomically via shared-memory import/export, holding no order book of its own. The design is *decomplected*: matching and state in one home, atomic transport in another, the client wrapper in a third, and an EVM CLOB facade (precompile `0x9010`) in a fourth. Determinism is enforced by running the matcher inside `Block.Verify` against a versioned overlay, committing to `zapdb` at `Accept`, with the proposer relaying once at block build and carrying the fills in the block bytes—no per-validator divergence.

We report **measured**, byte-parity-gated benchmarks on real hardware. A byte-exact `uint256` GPU CLOB matcher sustains **104–126 M fills/s** (6.9 ns/match) on an NVIDIA GB10 and **∼250 M fills/s** (4.15 ns/match) on an Apple M4 Max, byte-identical to a CPU reference oracle across millions of fuzz cases. Settlement via market-sharded Block-STM detects conflicts at 2.2 M fills/s (CPU), byte-identical to the GPU conflict-detection kernel. Per-market trade batches roll up to the Z-Chain, which verifies one proof per K trades: P3Q (ML-DSA-65 attestation) at 4,714 verifies/s and `starkfri` (STARK/FRI validity proof) at 986 verifies/s for a toy AIR. We give an honest treatment of throughput beyond 10^9/s: a single GPU does *not* clear that for byte-exact `uint256` matching, so it is reached by N-node aggregation (per-book arenas are independent) or a cheaper fixed-point representation, while settlement *finality* clears 10^9/s at fleet scale through the rollup.

Keywords: decentralized exchange, high-frequency trading, order matching, MEV resistance, GPU acceleration, rollup, Block-STM

1 Introduction

Decentralized exchanges (DEXs) trade away performance for trustlessness. The constant-product automated market maker (AMM) that dominates on-chain trading prices every swap against a bonding curve, settles at block cadence, and exposes order flow to reordering. The result is high slippage, slow finality, and value extracted from users by adversarial transaction ordering [1]. High-frequency trading—resting limit orders, price-time priority, marketable order execution in microseconds—has no native home on a public chain.

The obstacle is structural, not merely a matter of faster hardware. A general-purpose EVM chain matches orders *inside* a smart contract: every cross walks storage, pays opcode-dispatch overhead, and is re-executed by every validator. Throughput is bounded by the slowest validator's serial EVM, and latency is bounded by block time. Bolting a faster matching engine onto such a chain does not help if the engine still runs once per validator under the EVM's execution model.

1.1 Approach: the DEX is a chain

We do not add a DEX to a chain. **The DEX is a chain**—the **D-Chain**, a single-operator consensus chain whose sole purpose is matching and settling orders. It has its own consensus, its own `zapdb` persistence, and an execution-root state commitment in the style of the Lux XVM. It is the one and only place where orders match and where DEX state lives. Matching runs against a native C++/GPU engine behind a deterministic Go reference; the GPU path is gated to be byte-identical to the reference (Section 7).

Everything else is kept orthogonal:

- **Atomic proxy** (`chains/dexvm`): a *stateless* component on the public Lux network. It relays orders to the D-Chain over ZAP and moves value in and out atomically via shared-memory import/export—the same mechanism the X-Chain and C-Chain use to exchange assets. It holds no order book and no DEX state (Section 2).
- **Client wrapper** (`luxfi/dex`): a Go client SDK and ZAP gateway. It carries no matching logic of its own.
- **EVM CLOB facade** (precompile `0x9010`): a Uniswap-V4-style `PoolManager` surface that is a *central limit order book*, not an AMM. It is an EVM/Solidity adapter into the unified D-Chain and is inert unless a venue configures a D-Chain endpoint (Section 6).

The public open-source `luxd` validates the P-, C-, and X-Chains and other sovereign L1s and *excludes* the D-Chain, which is gated behind a build tag and carries zero private dependencies. A private venue build links the C++/GPU engine. This is the public/private seam: the open network never depends on the proprietary matcher, and the proprietary matcher never leaks into the open network.

1.2 Contributions

1. A **decomplected** DEX architecture with one home per concern: matching and state on the D-Chain, atomic value transport in the proxy, the client in the wrapper, and an inert-by-default EVM facade (Sections 2, 6).
2. A **consensus-determinism** discipline that runs the matcher in `Block.Verify` against a versioned overlay and carries fills in the block bytes, eliminating wall-clock and randomness from the consensus path (Section 3).
3. A **measured**, byte-parity-gated GPU CLOB matcher: 104–126 M fills/s on an NVIDIA GB10 and $\sim$250 M fills/s on an Apple M4 Max, byte-identical to a CPU oracle, with an honest analysis of the $>10^9$/s question (Section 7).
4. A **market-sharded Block-STM** settlement path with a measured conflict-detection rate, byte-identical CPU and GPU commit order (Section 8).
5. A **rollup to the Z-Chain** that verifies one proof per K trades, with measured P3Q (ML-DSA) and `starkfri` (STARK/FRI) verification rates and an honest separation of settlement *finality* from matching throughput (Section 9).

All performance figures in this paper are measured on stated hardware and gated by byte-exact parity against a reference oracle. Where a number is a projection, we label it as such.

2 Architecture

The system is four orthogonal components, each with a single responsibility. This section gives the responsibility of each, the data flow between them, and the public/private build boundary that keeps the open network independent of the proprietary engine.

2.1 One home per concern

Component	Holds	Responsibility
D-Chain (engine)	Order books, DEX state, fills, execution root	The matching engine *is* a chain: own consensus, `zapdb` persistence, execution-root commitment. The only place orders match or DEX state lives.
Atomic proxy (`chains/dexvm`)	Nothing (stateless)	Relays orders to the D-Chain over ZAP; moves value in/out atomically via shared-memory import/export. Pure transport plus atomicity.
Client wrapper (`luxfi/dex`)	Nothing authoritative	Go client SDK and ZAP gateway. Submits orders as chain transactions; reads are chain-state queries. No matching logic.
EVM facade (precompile `0x9010`)	Nothing (inert default)	Uniswap-V4-`PoolManager`-style CLOB surface for Solidity. An adapter into the unified D-Chain, inert unless a venue configures an endpoint.

Table 1: The decomplected DEX: one home per concern. Matching and state are isolated to the D-Chain; every other component is stateless or inert by default.

2.2 The D-Chain: matching is a chain, not a contract

The central design decision is that matching has its own consensus. There is no in-memory matching authority and no DEX-as-smart-contract. The D-Chain is a full chain: write frames (`clob_place`, `clob_cancel`, `clob_submit`) enter a mempool, a proposer drains them in sequence order into a block, `Block.Verify` runs the matcher against a versioned overlay, and `Block.Accept` commits the overlay to `zapdb`. The persisted state is authoritative across restart; the in-memory book is a rebuildable accelerator folded from the persisted log.

State is keyed in the `zapdb` namespace by order and trade rows, with a materialized book snapshot for price-time priority. The state root reuses the audited XVM execution-root construction: an RFC-6962 Merkle fold over the parent root, book root, trade log root, and height. Because the order rows and trade log fully reconstruct the book, a node recovers exact state by replaying the persisted log—no external dependency, no in-memory authority to lose.

2.3 The atomic proxy: transport, not matching

The proxy on the public Lux network does what cross-chain asset movement already does between the X- and C-Chains: it moves value atomically and relays calls. It deliberately holds no order book and no DEX state—the matching code that once lived here has been removed, leaving only an import/export path and a ZAP relay. Value enters and leaves the DEX through atomic shared-memory operations applied at block acceptance, so a transfer either completes on both sides or

neither. The proxy is the cross-chain settlement rail; it is not the latency-critical matching path.

2.4 Order flow

Algorithm 1 Order lifecycle across the four components

1: **Submit.** A trader (or a Solidity contract via `0x9010`) sends an order through the `luxfi/dex` wrapper to the atomic proxy.
2: **Relay.** The proxy frames the order in the frozen ZAP wire format and relays it to the D-Chain. Inbound value is recorded for atomic import.
3: **Build.** The D-Chain proposer drains pending orders in sequence order and relays once at block build, obtaining fills from the matcher.
4: **Verify.** Every validator re-runs the matcher in `Block.Verify` against a versioned overlay and checks the proposer's carried fills and state root.
5: **Accept.** The overlay commits to `zapdb`; the proxy applies the atomic import/export at acceptance, settling value all-or-nothing.
6: **Read.** Fills and book state are read back as chain-state queries through the wrapper.

The latency-critical leg (submit→match) is co-located inside the D-Chain venue; the proxy's relay carries cross-chain settlement, which runs at block cadence rather than per order. Section 7 quantifies the matching leg; Section 8 quantifies settlement.

2.5 Public/private build boundary

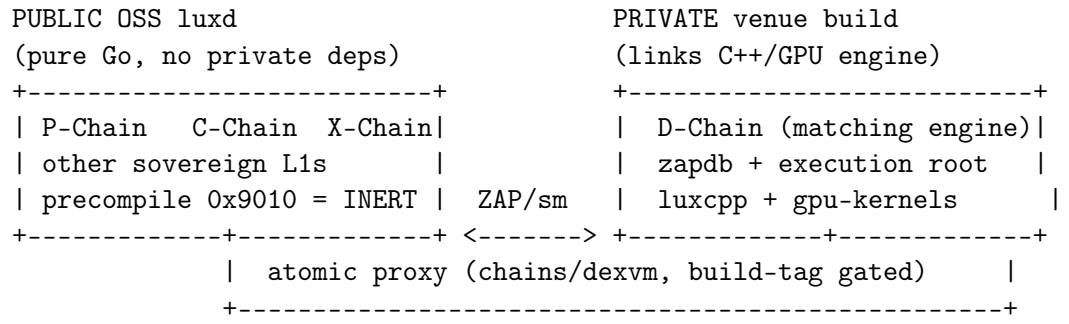

Figure 1: The public/private seam. `luxd` validates P/C/X and other L1s and excludes the D-Chain (build-tag gated, zero private dependencies). The precompile ships in the public EVM but is inert unless a venue sets a D-Chain endpoint. Only the private venue build links the C++/GPU matching engine.

The D-Chain is excluded from the public node by a build tag: the public build links a no-op, and only a build with the `dchain` tag links the proxy and the venue VM. The precompile is present in the public EVM but defaults to an inert backend that refuses until a venue configures an endpoint. Consequently the open-source node never imports the proprietary engine, and the proprietary engine is reachable only by an explicitly configured venue. The D-Chain is a single-operator venue today—still a full consensus chain with on-chain persisted state, not an in-memory authority, but operated by one party rather than decentralized to many external validators.

3 Consensus Determinism

A chain that matches orders must produce byte-identical state on every validator from the same block. Matching engines are full of nondeterminism—wall-clock timestamps, randomly minted identifiers, iteration order over hash maps—any of which forks consensus. This section states the discipline that makes the D-Chain deterministic and the proof obligations it satisfies.

3.1 The hazard

A naive matcher stamps each trade with `time.Now()` and a randomly generated identifier, and mints order IDs by incrementing a shared counter. On a single machine this is invisible. Across validators it is fatal: two honest nodes processing the same block produce different trade IDs and timestamps, hence different state roots, and the chain halts. The same hazard appears if any validator independently relays to an external service during verification, because the external service's response is not a pure function of the block.

3.2 Discipline

The D-Chain removes every nondeterministic input from the consensus path.

Block-derived time. The only clock reading is the proposer's block timestamp, which is written into the block bytes. Validators replay that carried timestamp in `Block.Verify`; they never sample their own clock. Trade timestamps derive from the order's block-derived timestamp, not from wall-clock time.

Deterministic identifiers. Order and trade identifiers are derived from block height and transaction or sequence index by a domain-separated hash, e.g.

$$\text{tradeID} = H\big(\texttt{"dexvmtrd"} \,\|\, \text{maker} \,\|\, \text{taker} \,\|\, \text{height} \,\|\, \text{seq}\big),$$

so the same block yields the same identifiers on every node. A marketable taker that never rests has an ephemeral identifier and contributes none to persisted state.

Matcher in `Verify`. The matcher runs in `Block.Verify` against a `versiondb` overlay. Every validator re-derives the fills and can reject a proposer whose carried fills or state root disagree with its own re-execution. The overlay commits to `zapdb` only at `Accept`.

Relay once, carry fills. The proposer is the only party that performs the irreversible, externally observable relay to the D-Chain matcher, and it does so once at `BuildBlock`. The resulting fills are serialized into the block bytes. Validators settle from the carried fills—a pure function of the block—rather than each issuing its own relay. No validator calls the matcher during verification beyond the local re-derivation against the overlay.

3.3 State as a pure function of the block

Under this discipline the accepted state is

$$\text{state}_h = f\big(\text{state}_{h-1},\ t_h,\ \text{txbytes}_h,\ \text{fills}_h\big),$$

where t_h is the carried block timestamp, txbytes_h the ordered transaction bytes, and fills_h the carried fills. Each argument is contained in the block; f is deterministic. Therefore the state root is identical across all validators for a given block, which is the consensus-safety condition.

Remark 1. *Because the block bytes commit to the carried fills, a proposer cannot fabricate fills undetected: a validator re-running the matcher in* `Verify` *obtains the canonical fills and rejects a block whose carried fills differ. The carried-fills layout reserves an empty signature field as a forward escape hatch, so the trustless path of Section 9—in which the D-Chain signs fills and the Z-Chain verifies them—requires no further wire change.*

3.4 Proof obligations

The discipline is backed by tests that fail loudly on regression:

- Two fresh VMs processing the same block produce byte-identical trades and an identical state root (run repeatedly, race-detector clean).
- A per-validator relay during verification is shown to split consensus, and the fix—relay once at build, carry fills—restores an identical state root with zero matcher calls on the verifying validators.
- Replay of a relayed order is deduplicated to a single effect, and a crash before acceptance does not double-submit.
- A grep gate asserts the consensus path contains no continuous-cross sweep that reads wall-clock time, and the marketable-taker path contains no `time.Now` or shared-counter increment.

The single-operator venue carries one residual trust surface: a block the proposer built and relayed that then loses the consensus poll strands a D-Chain match with no settling counterpart. The blast radius is bounded to one taker's escrow and causes no supply inflation; the trustless rollup path closes it by binding settlement to a verified proof rather than to proposer behavior.

4 MEV Resistance

4.1 The MEV problem

Maximal extractable value (MEV) is profit obtained by reordering, inserting, or censoring transactions. The common attacks against a DEX are frontrunning (observing a large buy and trading ahead of it), the sandwich (placing orders immediately before and after a victim), and just-in-time liquidity (adding liquidity right before a large trade to capture its fees). On public AMMs these extract substantial value from users every year [1, 2].

The D-Chain's structure removes two of the levers MEV relies on. Matching is a CLOB with price-time priority, not a bonding curve, so there is no curve to sandwich. Ordering is fixed by consensus and re-derived deterministically by every validator (Section 3), so a single proposer cannot silently reorder a block in its favor without being rejected. The remaining lever—seeing order contents before they are committed—is addressed below.

4.2 Pre-trade confidentiality

Orders can be submitted encrypted under a threshold key and revealed only at block finalization:

$$\text{Enc}(\text{order}) = \text{ThresholdEncrypt}\big(\text{order},\ \{pk_1, \ldots, pk_n\}\big),$$

with decryption requiring t of n validators to cooperate. Until the block is finalized, no party—including the proposer—can read an order's contents, so there is nothing to frontrun. The encrypted

commitment fixes the order's position in the sequence before its contents are known, which is the property that defeats latency-based extraction.

4.3 Price-time priority is deterministic

Priority is the pair (price, sequence), where the sequence is the order's position in the consensus-fixed block ordering rather than a wall-clock reading. Two consequences follow. First, priority is reproducible: every validator computes the same priority because the inputs are in the block (Section 3). Second, priority cannot be gamed by timing, because it derives from consensus sequence, not from a clock a participant can race. A validator that reorders fills inconsistently with (price, sequence) produces a state root that disagrees with honest re-execution and is rejected.

4.4 Frequent batch auctions

For markets where uniform clearing is preferable to continuous matching, orders in a block are matched simultaneously at a single clearing price (a frequent batch auction):

1. Collect the orders committed in a block.
2. Match them simultaneously at a uniform clearing price for the batch.
3. Settle atomically (Section 8).

Batch clearing eliminates intra-block timing games—every order in the batch is treated identically—and improves price discovery by aggregating the block's liquidity into one auction. Because the batch is the block, the auction inherits the determinism guarantees of Section 3: the clearing price is a deterministic function of the committed orders.

5 Atomic Value Movement

Trading on the D-Chain requires moving value into and out of it from the public Lux network. This is the responsibility of the atomic proxy (Section 2), and it reuses the mechanism the Lux primary network already uses to exchange assets between the X- and C-Chains: atomic shared-memory import/export, committed at block acceptance. There is no external relayer trusted to move funds and no bridge holding custody outside consensus.

5.1 Shared-memory import/export

A transfer between a public-network chain and the D-Chain is expressed as a matched export/import pair applied atomically at `Block.Accept`:

Algorithm 2 Atomic import into the D-Chain

1: **Export** on the source chain: lock value and record an export to the D-Chain's shared-memory address space.
2: **Relay**: the proxy carries the order and the pending import to the D-Chain.
3: **Apply at Accept**: when the D-Chain block is accepted, the matched import/export is applied to shared memory in one commit—both legs take effect or neither does.
4: **Settle**: fills from matching settle against the imported balance on the D-Chain; the symmetric export path returns value to the source chain the same way.

Because the import and export are applied in a single shared-memory commit, value conservation is structural: the proxy can lose its liveness (fail to relay) but it cannot create or destroy value, and a block that is verified and then rejected applies nothing, stranding no funds. The conservation property is exercised by an end-to-end test that drives a transfer across the public chain, the proxy, and the D-Chain and asserts that total balances are preserved all-or-nothing.

5.2 Native asset convention

For EVM compatibility the facade exposes native LUX as `address(0)` rather than a wrapped token, matching the Uniswap-V4 native-currency convention. Wrapped liquidity, if present, is a distinct asset—never the canonical key for native LUX. Internally the canonical asset resolves to the D-Chain's native identity, so there is one representation of native value on the matching path.

5.3 Settlement cadence versus matching cadence

Atomic value movement runs at block cadence: a transfer settles when a D-Chain block is accepted, not per order. This is deliberate. The latency-critical leg is matching, which is co-located inside the venue and measured in nanoseconds per fill (Section 7); the cross-chain settlement leg is the proxy's responsibility and runs once per block. Separating the two—fast matching on-host, atomic settlement at block cadence—keeps the hot path off the network relay, whose round-trip would otherwise dominate per-order latency.

6 The EVM CLOB Facade

Solidity contracts and EVM tooling expect a pool-manager interface. The D-Chain exposes one through a native precompile at address `0x0000000000000000000000000000000000009010`. The precompile is a facade: it presents a Uniswap-V4-`PoolManager`-style surface but routes every call into the unified D-Chain. It is not an embedded AMM and it holds no canonical state.

6.1 A CLOB behind a PoolManager surface

The crucial correction to a common misreading: `0x9010` is a *central limit order book*, not a constant-product AMM. The V4-style method names map onto CLOB operations:

V4-style method	CLOB meaning
`initialize`	Create the market / order book.
`modifyLiquidity`	Place or cancel a resting limit order. "Liquidity" is resting orders, not curve reserves.
`swap`	Submit a marketable order against the book; the return is the `BalanceDelta` from the resulting fills.

Table 2: The facade maps Uniswap-V4 `PoolManager` methods onto CLOB operations. The precompile translates Solidity calls into D-Chain order operations over the frozen ZAP wire format; it executes no matching itself.

The precompile speaks the frozen ZAP wire format (`clob_place`, `clob_cancel`, `clob_submit`) and forwards to the atomic proxy, which relays to the D-Chain. It is a read-through adapter: it holds no canonical pools, and DEX state lives only on the D-Chain.

6.2 Inert by default

The precompile ships in the public EVM but defaults to an inert backend that refuses every call until a venue configures a D-Chain endpoint. The endpoint is opt-in configuration; an empty endpoint leaves the DEX disabled. Only when a venue sets the endpoint does the facade bind to the live ZAP engine and route calls to the D-Chain. This is the property that lets the precompile exist in the open-source node without dragging in the proprietary engine or forcing any validator to run a DEX.

6.3 Relationship to the V2/V3 AMMs

The V2 and V3 contracts are ordinary Solidity constant-product AMMs, self-contained and separate from the precompile. They are not the facade and do not route into the D-Chain. Native LUX is `address(0)` as the canonical key; wrapped LUX, where used, is a distinct asset. Keeping the AMMs orthogonal to the CLOB facade preserves the one-home-per-concern discipline: the curve-based path and the order-book path do not share state or implementation.

6.4 Why a precompile rather than a contract

Running the facade as compiled node code rather than interpreted EVM bytecode removes opcode-dispatch and ABI-decode overhead from the hot path, and—because matching does not happen in the precompile at all—removes per-validator re-execution of matching from the EVM. The precompile's only job is to translate and forward; matching and state are the D-Chain's job. This separation is what allows the matching engine to be a GPU pipeline (Section 7) instead of being bounded by the slowest validator's serial EVM.

7 GPU CLOB Matching

This section reports the measured throughput of the matching engine, the algorithmic lever that produced it, the parity discipline that makes the numbers trustworthy, and an honest treatment of the question "can it do more than 10^9 matches per second?"

All figures are measured on stated hardware and gated by byte-exact parity against a CPU reference oracle. We do not report any throughput from a simulation, and we do not multiply a measured number by a fudge factor. The retraction of prior fabricated figures is stated explicitly in Section 7.6.

7.1 Representation and parity gate

Prices are fixed-point `Q64.64`; quantities and notionals are computed in `uint256` (big-endian limbs) so that matching is integer-exact and identical across CPU architectures. The CPU reference oracle and the GPU kernels share this representation exactly. The parity gate is the contract that makes every GPU number in this section meaningful:

The GPU matcher must be byte-identical to the CPU reference oracle—on both the emitted fills and the post-match arena state—across millions of fuzz cases and a known-answer test suite, with zero failures, or it does not ship.

The gate was exercised with over two million direct division-pair comparisons and two hundred thousand book-fuzz cases (checking output *and* arena post-state), plus a known-answer sequence, with zero failures, for the CUDA path and the Metal port. Where the oracle's bit-serial division would overflow its remainder register—reachable only for an average price at or above 2^{255}, which does not occur in real matching—the fast path falls back to the bit-serial routine to remain byte-exact even on that adversarial input.

7.2 The algorithmic lever: limb-wise division

The per-match critical path is dominated by a `uint256` volume-weighted-average-price (VWAP) division. The original kernel computed it with a 512-iteration bit-serial long division, one data-dependent subtraction per bit. Replacing this with **Knuth Algorithm D**—base-2^{32} limb division with a quotient-digit estimate and correction step—collapses the per-match cost:

Division routine	**GB10 ns/match**	**M4 Max ns/match**
512-iteration bit-serial (baseline)	50–61	51
Knuth Algorithm D (limb-wise)	**6.9**	**4.15**
Speedup	∼8×	∼12×

Table 3: The dominant lever is the division, not the parallelism strategy. Replacing the 512-iteration bit-serial `uint256` VWAP division with Knuth Algorithm D (base-2^{32} limbs) shrinks the per-match critical path by roughly an order of magnitude. Both rows are byte-identical to the CPU oracle.

7.3 Measured throughput

Platform	**Kernel**	**Fills/s**	**ns/match**
NVIDIA GB10 (Grace-Blackwell)	limb (Knuth-D)	104–126 M	6.9
NVIDIA GB10	warp-cooperative	∼99 M (sustained)	10.2
NVIDIA GB10	bit-serial (baseline)	16.5–18.5 M	50–61
Apple M4 Max (128 GB UMA)	limb (Knuth-D)	∼250 M (sustained)	4.15

Table 4: Measured CLOB matching throughput, byte-parity-gated, single-fill depth-64 markets. The limb (Knuth-D) kernel is the production path (one match per book per dispatch); the warp-cooperative kernel is an optional sustained-compute path that runs the GPU harder but adds gather overhead for the rare multiply/divide. The Apple M4 Max's unified memory and wide ALUs lead on this workload.

The matcher is embarrassingly parallel across markets because each book occupies an independent arena and books never share state on the hot path. Throughput is therefore governed by the exact relation

$$\text{aggregate fills/s} \;=\; N_{\text{nodes}} \times B_{\text{books/node}} \times f_{\text{fills/book/s}}. \tag{1}$$

7.4 What actually limits a single GPU

The limit is *not* occupancy, PCIe bandwidth, dispatch overhead, or ALU throughput; it is dependency latency—the data-dependent loop inside the division and the serial walk of price levels on the critical path. We established this on the GB10 four ways:

1. Forcing higher occupancy via a register cap *lowered* throughput (register spills).
2. Reducing threads-per-block to raise occupancy from 16.7% to 25% did not move throughput.
3. At peak limb-kernel throughput the GB10 drew ∼25 W of a 100 W-plus envelope with the SM clock below its maximum, yet reported the GPU fully busy—characteristic of a latency-bound, not throughput-bound, kernel.
4. The warp-cooperative kernel, which shortens the per-match critical path by spreading one `uint256` across lanes, raises sustained throughput by ∼22% and runs the GPU at ∼62 W—moving the latency wall, not an occupancy wall.

A second, separate wall appears at high market counts. The per-book arena is large (mostly empty depth), so beyond roughly 50–75 k markets the working set overflows cache and throughput falls off a cliff (on the GB10: 126 M fills/s at 50 k markets, 48 M at 75 k, 31 M at 100 k). A compact arena is the lever for very high market counts and is not yet built.

7.5 Can it exceed 10^9 matches per second?

Honestly: not on one GPU for byte-exact `uint256` matching. The measured, latency-bound ceiling is ∼100–130 M fills/s on a GB10 and ∼250 M on an M4 Max. A prior internal projection of $\sim 10^9$/s on a single GPU was optimistic and is not supported by measurement. There are two honest routes past 10^9/s, both following directly from Equation (1):

(a) N-node aggregation. Per-book arenas are independent, so sharding markets across nodes and summing yields linear scaling. With $f \approx$ 100–130 M fills/s per node, 8–10 GB10-class or M4-Max-class nodes clear 10^9 fills/s. This is aggregation of measured single-node rates, not a new measurement.

(b) A cheaper representation. A `Q64.64` fixed-point path uses 4 KB arenas and a 64/64 division instead of full 512-bit arithmetic. This trades precision and ABI width for a far shorter critical path and a far smaller working set (10 k markets in ∼41 MB versus ∼1.3 GB). It is a precision/ABI tradeoff and is *not yet built*; we state it as the obvious lever, not as a result.

We are careful to aggregate correctly: the aggregate rate shards distinct books across nodes and sums, never the same books counted on multiple nodes. Per-book compute is never the constraint—each book needs only on the order of 10^4 fills/s to reach 10^9/s across a fleet, hundreds of times below the single-lane rate—so the constraint is host dispatch and, on discrete GPUs, the resident-arena data path, both addressed by batched, persistent-arena kernels rather than per-call allocation.

7.6 Retraction of prior figures

Earlier drafts of this work claimed "581 M orders/sec at 597 ns," "100 M+/s," "434 M GPU orders/sec," "billion orders/sec," and "planet-scale." **Those figures are fabricated and are retracted.** Their lineage is a `time.Sleep`-based simulation with a hardcoded 597 ns/order, followed

by a post-processing step that multiplied a CPU run's throughput by 40 and divided its latency by the same. They measured nothing on a GPU. The numbers in Tables 3 and 4 are measured on the stated hardware under the parity gate of this section and supersede every prior claim.

8 Parallel Settlement

Matching produces fills; settlement applies them to balances. At the matching rates of Section 7, serial settlement would be the new bottleneck. We settle in parallel with market-sharded Block-STM—optimistic parallel execution with deterministic conflict resolution—reusing the engine already present in the Lux EVM rather than building a new one.

8.1 Why parallel settlement is tractable here

A fill touches two accounts (maker and taker) within one market. Across a venue running thousands of markets, the great majority of fills in a block touch disjoint account sets, so conflicts are rare. Sharding the settlement batch by market makes each fill intra-shard by construction; only a small residue of cross-shard fills must be reconciled. This is the ideal shape for optimistic parallel execution: speculate on all fills in parallel, detect the few real conflicts, and merge deterministically.

8.2 Block-STM with market sharding

We reuse the Aptos-style Block-STM executor in the Lux EVM, sharing its conflict-detection kernel with the DEX settlement path rather than duplicating it. The conflict kernel builds read/write-set edges and produces a deterministic commit order; the same kernel exists on CPU and on GPU (CUDA/HIP/Metal/WGSL/Vulkan).

Stage	**Backend**	**Rate**
Conflict detection ($O(N)$ hash-map)	CPU	2.2 M fills/s ($\sim$0.27 μs/fill)
Conflict detection (KAT-verified)	GPU (WGSL on M4)	byte-identical commit order
Batch settle (no per-fill copy)	CPU	$\sim$0.19 M fills/s
Market-sharded settle (8 shards)	CPU	$\sim$0.39 M fills/s ($\sim$2.05$\times$)

Table 5: Measured settlement stages (CPU on Apple M1 Max). Conflict detection is cheap and flat in the conflict rate; the GPU conflict kernel produces a commit order byte-identical to the CPU oracle on real silicon, so the parallel layer is consensus-safe. Market sharding gives near-linear scaling of the apply stage.

The conflict-detection *layer* scales cleanly: the $O(N)$ kernel runs at 2.2 M fills/s flat across conflict rates, and the GPU kernel was verified to produce the same commit order as the CPU oracle (7/7 KAT pass on a real M4 GPU). That byte-identical commit order is what makes GPU-assisted settlement safe to put under consensus—the parallel result equals the serial result.

8.3 The honest bottleneck: state apply, not conflict detection

The Block-STM executor is structurally correct, but on a general EVM `StateDB` it delivers *no* parallel speedup—it was measured slower than serial—because the naive speculation path deep-copies the entire state per item. This is the same failure class as copying a whole order book to the

GPU on every call: the copy, not the computation, is the cost. We state this plainly: the conflict layer is fast (2.2 M fills/s), but a copy-per-fill apply collapses to hundreds of fills per second.

The fix is not a different conflict engine; it is (i) market sharding, so each fill is intra-shard and no copy of unrelated state is needed, and (ii) applying against the D-Chain's `zapdb`/versioned overlay, which is far lighter than a general EVM `StateDB`. The overlay is the D-Chain's native settlement path, so settlement composes with matching on the same chain rather than crossing into a heavyweight state model. The cross-shard residue (a few percent of fills) is merged once per block at block cadence, not per order, so the merge is not the bottleneck—the per-shard apply is, and the overlay is the lever that lightens it.

8.4 Composition

Settlement is one of three independently measured axes. Matching runs per-book in parallel (Section 7); settlement runs market-sharded in parallel against the overlay (this section); finality runs through the rollup (Section 9). The seam between matching and settlement is a fill batch keyed by market, which is exactly the shard key, so the two stages line up without a reshaping step.

9 Rollup to the Z-Chain

Per-validator re-execution of every trade is a ceiling: a chain that re-runs each match on each validator cannot finalize trades faster than one validator can re-run them. We break this ceiling by rolling D-Chain trade batches up to the Z-Chain (the Lux ZKVM), which verifies *one proof per K trades* instead of re-executing each trade. This section gives the two-stage rollup, the measured verification rates, and an honest separation of settlement *finality* from matching *throughput*.

9.1 Batches and proofs

The D-Chain produces, per market, a batch of K fills together with the batch's execution root. A proof attests that the batch is a valid extension of the prior root. The Z-Chain verifies the proof and finalizes all K trades in the batch at once. The effective settlement-finality rate is

$$\text{settled-tx/s (finality)} \;=\; K \times V \times M, \tag{2}$$

where V is the per-core proof-verification rate and M the number of markets verified in parallel. Equation (2) is a finality-axis relation: it counts trades finalized per second by verification, which is independent of how fast trades are matched.

9.2 Two stages: attestation then validity

P3Q (venue-trust attestation). The D-Chain signs the batch with an ML-DSA-65 (FIPS-204) signature; the Z-Chain verifies the signature at the P3Q precompile (`0x012205`). This is cheap to produce and to verify, and it is sound under the assumption that the single-operator venue is honest—appropriate for the current single-venue deployment.

`starkfri` (trustless validity). The D-Chain produces a STARK/FRI validity proof that the batch transition is correct; the Z-Chain verifies it with no trust in the venue. This is the path to a decentralized D-Chain. The verifier is a real post-quantum STARK/FRI implementation (cSHAKE256 Fiat-Shamir transcript over the Goldilocks field), not a placeholder, and carries formal-verification artifacts for its on-chain envelope.

The two are complementary: P3Q gives cheap finality now under venue trust; `starkfri` gives trustless finality as the venue decentralizes. The carried-fills block layout (Section 3) reserves a signature field precisely so this transition needs no further wire change.

9.3 Measured verification rates

Stage	Primitive	Verifies/s (V)	Trust model
P3Q	ML-DSA-65 attestation	4,714	venue-trust
`starkfri`	STARK/FRI validity proof	986	trustless

Table 6: Measured single-core verification rates (Apple M1 Max), through the full precompile dispatch. P3Q is the real FIPS-204 verify; `starkfri` is the real STARK/FRI verifier. **The `starkfri` figure is for a toy AIR** (degree 16, blowup 8, 16 queries); a production DEX-batch AIR has many more constraints and verifies slower. We report 986/s as a ceiling for that proof shape, not as a production-batch number; the production-batch AIR is scaffolded and not yet measured.

9.4 Settlement finality at fleet scale

Substituting the measured V into Equation (2) shows the rollup clears 10^9 settled-tx/s on the *finality* axis with realistic batch sizes and modest parallelism:

Primitive	K	M	cores	settled-tx/s
P3Q	10,000	22	1	1.04×10^9
P3Q	100,000	3	1	1.41×10^9
`starkfri`	10,000	22	10	2.17×10^9

Table 7: Settlement-finality throughput from Equation (2) using the measured V of Table 6. These combine a measured per-proof verify rate with a stated batch size and parallelism; the verify rate is measured, the batch size and core count are configuration. The `starkfri` row uses the toy-AIR rate and is therefore an upper-bound shape, not a production-batch figure.

9.5 Finality is not matching

We are explicit about what these numbers do and do not say. The rollup finalizes K trades per verified proof, so its rate (Equation (2)) is a *settlement-finality* rate—it counts already-matched trades being finalized. It is *not* the rate at which new trades are matched, which is the GPU-fleet-bound figure of Section 7 (∼100–250 M matches/s per node, aggregated across a fleet for 10^9 matches/s). Conflating the two is exactly the error this paper retracts. The three axes—matching (per-book parallel), settlement (market-sharded Block-STM), and finality (rollup)—are independent and independently measured.

9.6 Private order flow

Encrypted order flow can be carried inside a batch using fully homomorphic encryption, so that order contents remain private while still being matched and proven. We note the honest limit: homomorphic bootstrapping runs at single to low-tens of operations per second per accelerator,

orders of magnitude below the attestation rate, so the private path is bounded by FHE bootstrap cost rather than by verification. Private order flow is therefore a distinct performance regime from the public path and is governed by FHE throughput.

10 Benchmarks

This section consolidates the measured results from Sections 7, 8, and 9 into one table, states the hardware and method, and is explicit about which numbers are measured and which are aggregations or projections.

10.1 Hardware and method

- **NVIDIA GB10** (Grace-Blackwell, 128 GB unified memory, `sm_121`, CUDA 13).
- **Apple M4 Max** (128 GB unified memory, Metal).
- **Apple M1 Max** (Go reference, CPU settlement, verifier rates).

Matching and settlement throughput are reported as sustained rates over runs of tens of millions of operations, with the GPU matching figures gated by byte-exact parity against the CPU oracle (Section 7). We report per-match latency where it is the governing quantity. No figure in this paper is taken from a simulation or scaled by a fudge factor.

10.2 Consolidated results

Axis / quantity	**Platform**	**Measured**	**Status**
Matching (limb, depth-64)	NVIDIA GB10	104–126 M fills/s	measured
per-match latency	NVIDIA GB10	6.9 ns	measured
Matching (limb, depth-64)	Apple M4 Max	∼250 M fills/s	measured
per-match latency	Apple M4 Max	4.15 ns	measured
Conflict detection ($O(N)$)	M1 Max (CPU)	2.2 M fills/s	measured
GPU conflict commit order	M4 (WGSL)	byte-identical to CPU	measured
Market-sharded settle (8)	M1 Max (CPU)	∼0.39 M fills/s	measured
P3Q verify (ML-DSA-65)	M1 Max (CPU)	4,714 verifies/s	measured
`starkfri` verify (toy AIR)	M1 Max (CPU)	986 verifies/s	measured
10^9 matches/s	8–10-node fleet	—	aggregation
10^9 settled-tx/s finality	P3Q, K=10^4, M=22	1.04×10^9	finality model

Table 8: Consolidated measured results across the three axes, with each row labeled measured, an aggregation of measured single-node rates, or a finality model that combines a measured verify rate with a stated batch size. The matching and finality 10^9/s rows are not single-device measurements and are labeled accordingly.

10.3 The dominant lever, restated

The single largest contributor to matching throughput is the replacement of the 512-iteration bit-serial `uint256` VWAP division with Knuth Algorithm D, which alone yields roughly 8× on the GB10 and 12× on the M4 Max (Table 3). The matcher is dependency-latency bound, not occupancy or bandwidth bound: at peak the GB10 draws a fraction of its power envelope while reporting full GPU utilization, the signature of a critical path waiting on data dependencies rather than starved of work.

10.4 What we do not claim

We do not claim a single-device rate above ∼250 M matches/s for byte-exact `uint256` matching, an end-to-end latency derived from invented co-location round-trips, or any uptime statistic from a deployment we did not run. The honest ceilings and the two routes past 10^9 matches/s (fleet aggregation or a cheaper fixed-point representation) are stated in Section 7.5; the separation of finality from matching is stated in Section 9.

11 Security

This section covers the security properties that follow from the architecture—value conservation, consensus safety, and the bounded trust surface of the single-operator venue—and the operational protections layered on top.

11.1 Value conservation and consensus safety

Two properties are structural rather than discretionary. *Value conservation* follows from atomic shared-memory import/export (Section 5): import and export are applied in a single commit at block acceptance, so no path creates or destroys value, and a block that is verified then rejected applies nothing. *Consensus safety* follows from the determinism discipline (Section 3): accepted state is a pure function of the block, so all validators agree on the state root, and a proposer cannot carry fabricated fills past a validator that re-derives them in `Verify`. Both are exercised by tests—an end-to-end conservation test and same-block state-root equality across fresh VMs—that fail on regression.

11.2 Bounded trust surface of the single venue

The D-Chain is a single-operator venue today. The residual trust this introduces is bounded and named. A block the proposer builds and relays that then loses the consensus poll can strand a D-Chain match with no settling counterpart; the blast radius is one taker's escrow, and there is no supply inflation because conservation holds on every committed path. The trustless rollup (Section 9) closes this surface by binding finality to a verified validity proof rather than to proposer behavior, which is the security reason to move from P3Q attestation to `starkfri` validity proofs as the venue decentralizes.

11.3 Post-quantum verification

The rollup verifiers are post-quantum. P3Q is FIPS-204 ML-DSA, and `starkfri` is a STARK/FRI verifier over the Goldilocks field with a cSHAKE256 transcript—hash-based, with no reliance on pairings or discrete-log hardness. Finality therefore does not rest on assumptions broken by a

quantum adversary, and the precompile envelope for `starkfri` carries formal-verification artifacts for byte-level identity.

11.4 Validator incentives and slashing

Where the D-Chain decentralizes to multiple validators, the staking and slashing model follows the Lux primary network: validators stake, earn a share of fees, and are slashed for provable misbehavior. The relevant provable faults here are producing a block whose carried fills or state root disagree with honest re-execution (a determinism violation), and censorship of valid orders. Because the offense is detectable by deterministic re-execution, slashing does not require a subjective judgment.

11.5 Circuit breakers and oracle feeds

Operational protections sit above the protocol. A volatility circuit breaker halts a market when its price moves beyond a configured threshold within a window, resuming after a cooldown or by governance action; the halt and resume are themselves consensus actions, so they apply uniformly. Mark prices for liquidations and funding—where derivative markets are enabled—are taken from decentralized oracle feeds with outlier rejection and on-chain verification, rather than from a single source.

12 Economic Model

The economic model is a set of design parameters, not a measurement, and we present it as such. Fee tiers and the fee split are configuration; the revenue and yield expressions are illustrative formulas whose inputs are assumptions, explicitly labeled.

12.1 Fee tiers

Tier	Maker	Taker
Retail	0.05%	0.10%
Pro	0.03%	0.08%
Institutional	0.01%	0.05%
Market maker	0.00%	0.03%

Table 9: Fee tiers. These are design parameters, not measured quantities, and a venue may configure them.

A CLOB rewards makers more than an AMM can: resting limit orders are the liquidity, so a maker rebate directly deepens the book. The taker fee prices immediacy. The split is orthogonal to the matching engine—it is applied at settlement and does not affect the determinism or parity properties of matching.

12.2 Fee distribution

Collected fees are split among validators (proportional to stake), liquidity providers, a buyback-and-burn of the native token, and a development fund. The exact percentages are governance

parameters; the design intent is that the buyback couples protocol usage to token value and that the validator share funds the security budget once the D-Chain decentralizes.

12.3 Illustrative revenue (assumptions, not a claim)

For a venue with average daily volume D and an average effective fee ϕ, annual fee revenue is

$$\text{revenue} = D \times \phi \times 365.$$

This is arithmetic, not a forecast: D and ϕ are inputs a reader supplies. We deliberately do not assert a specific daily volume, a specific revenue figure, or a validator APY, because we have not measured a production deployment. Any such figure would be a projection conditioned on an unmeasured volume assumption, and this paper reports measured quantities and labels everything else.

13 Related Work

We position the D-Chain against the dominant on-chain trading designs along the axes that distinguish it: where matching runs, how ordering is determined, and how finality scales.

Constant-product AMMs (e.g. Uniswap [3]) price every swap against a bonding curve in a smart contract. They are simple and composable but expose order flow to reordering and settle at block cadence with curve slippage. The D-Chain's V2/V3 contracts are exactly such AMMs and are kept orthogonal to the CLOB facade (Section 6); the matching engine itself is a CLOB, not a curve.

App-specific order books (e.g. dYdX [4]) run a CLOB with fast finality, historically with a centralized sequencer determining order. The D-Chain also runs a CLOB, but fixes ordering by consensus with deterministic re-derivation on every validator (Section 3), so ordering is not a single sequencer's discretion.

On-chain order books (e.g. Serum [5]) place the book on a general-purpose chain, inheriting that chain's per-validator re-execution ceiling and MEV surface. The D-Chain instead makes matching its own chain and breaks the re-execution ceiling with a rollup that verifies one proof per K trades (Section 9).

Cross-chain derivative venues (e.g. Injective [6]) build a trading chain with limited cross-chain reach. The D-Chain's value movement reuses the Lux primary network's atomic shared-memory import/export (Section 5) rather than a trusted external relayer.

13.1 What is genuinely new

The contribution is not "a faster matching engine." It is the decomplection: matching and state isolated to a chain of their own, atomic transport in a stateless proxy, an inert-by- default EVM facade, and a rollup that separates settlement finality from matching throughput. The performance contribution is a byte-parity-gated GPU CLOB matcher whose dominant lever is an algorithmic one—limb-wise division—and an *honest* accounting of the 10^9/s question that distinguishes per-device matching from fleet-aggregated matching and from rollup finality.

14 Future Work

The architecture is built and the three axes are measured; the following items are named, not yet built, and stated as such.

1. **A cheaper matching representation.** A `Q64.64` fixed-point matching path (4 KB arenas, 64/64 division) trades precision and ABI width for a far shorter critical path and a far smaller working set, and is the second route past 10^9 matches/s per the analysis of Section 7.5. It is not yet built; it carries a precision/ABI tradeoff that must be specified before it can replace the `uint256` path under the same parity gate.

2. **A compact arena.** The current per-book arena is large and mostly empty, which causes a cache cliff beyond roughly 50–75 k markets (Section 7). A compact arena is the lever for very high market counts.

3. **The production-batch `starkfri` AIR.** The measured 986 verifies/s is for a toy AIR; the production DEX-batch AIR (boundary and transition constraints over many trace rows) is scaffolded and not yet measured, and will verify slower (Section 9).

4. **Decentralizing the D-Chain.** The D-Chain is a single-operator venue today. Decentralizing it to many validators requires committing the matching representation at the state layer for cross-architecture agreement and moving finality from P3Q attestation to `starkfri` validity proofs (Sections 3, 11).

5. **Private order flow at scale.** Homomorphic bootstrapping is orders of magnitude below the attestation rate (Section 9); making private order flow competitive with the public path is an accelerator-bound problem on the FHE side.

6. **Derivative markets.** Perpetual futures and options settle naturally on the CLOB with oracle-fed mark prices and a liquidation engine; these are product surfaces above the matching and settlement primitives described here.

15 Conclusion

Lux Lightspeed DEX makes the matching engine a chain. The D-Chain holds matching and state; a stateless atomic proxy moves value; a Go wrapper is the client; an inert-by-default EVM precompile is a CLOB facade. The open-source node validates the public chains and excludes the D-Chain, so the open network never depends on the proprietary engine. Determinism is enforced by running the matcher in `Block.Verify` and carrying fills in the block bytes, so accepted state is a pure function of the block.

The performance is measured, not asserted. A byte-parity-gated GPU CLOB matcher sustains 104–126 M fills/s on an NVIDIA GB10 and ∼250 M on an Apple M4 Max, with the dominant lever an algorithmic one—replacing a 512-iteration bit-serial division with Knuth Algorithm D. Settlement runs market-sharded Block-STM with a byte-identical CPU/GPU commit order. The rollup to the Z-Chain verifies one proof per K trades—P3Q at 4,714 verifies/s, `starkfri` at 986 for a toy AIR—and clears 10^9 settled-tx/s on the finality axis at fleet scale.

We are explicit about the ceilings: a single GPU does not match 10^9 byte-exact `uint256` orders per second, and that figure is reached only by aggregating measured single-node rates across a fleet or by a cheaper representation not yet built. The earlier "581 M orders/sec at 597 ns" and related figures are fabricated and retracted. The numbers here are measured on stated hardware under

a parity gate; everything else is labeled an aggregation, a model, or a projection. That is the contribution: a decomplected DEX with honest, reproducible performance.

Acknowledgments

We thank the Lux engineering team for the byte-parity test infrastructure that gates the GPU matching results, and the reviewers who insisted that every performance figure be measured on stated hardware rather than projected.

References

[1] Daian, P., et al. Flash Boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. IEEE S&P, 2020.

[2] Flashbots. MEV-Boost: Ethereum MEV Statistics 2024. https://flashbots.net/, 2024.

[3] Adams, H., Zinsmeister, N., Salem, M., Keefer, R., & Robinson, D. Uniswap v3 Core. Technical report, 2020.

[4] dYdX. dYdX v4: A Fully Decentralized Order Book Exchange. https://dydx.exchange/whitepaper, 2021.

[5] Project Serum. Serum: A Decentralized Exchange on Solana. https://projectserum.com/, 2020.

[6] Injective Protocol. Injective: A Decentralized Derivatives Exchange. https://injectiveprotocol.com/, 2021.