LIQUIDITY.IO

Liquid DEX Performance

GPU-Accelerated Order Matching at 434 Million Orders Per Second

---

Zach Kelling

Satschel, Inc.

Version 1.0 — April 2026

434M Orders/Sec (GPU) · 2ns Latency · C++ / Metal / CUDA
FIX 4.4 Gateway: 6.16μs RTT · ZAP Protocol: 4.2M Calls/Sec
Native EVM Precompiles · Kansas City Datacenter · 200Gbps ConnectX-7

Contents

Abstract

This paper presents performance benchmarks for the Liquid DEX, the native order matching engine of the Liquidity Network (chain ID 8675309). By implementing the matching engine as GPU compute shaders (Metal on Apple Silicon, CUDA on NVIDIA) and exposing it through native EVM precompiles, the Liquid DEX achieves 434 million orders per second with 2ns per-order latency on GPU, and 1.01 million orders per second with 487ns latency on CPU. We detail the architecture, benchmark methodology, FIX 4.4 gateway performance, ZAP wire protocol throughput, and network infrastructure, comparing against NYSE Arca, NASDAQ, and Binance.

Introduction

Traditional exchange matching engines are CPU-bound, single-threaded processes optimized for sequential price-time priority matching. The fastest production systems—NYSE Arca at approximately 5 million orders per second and NASDAQ at approximately 10 million—represent the ceiling of single-threaded CPU architectures. Cryptocurrency exchanges operate at significantly lower throughput: Binance processes approximately 100,000 orders per second [9].

The Liquid DEX takes a fundamentally different approach. Rather than optimizing single-threaded CPU code, we reformulate order matching as a massively parallel GPU workload. A single order book is partitioned across thousands of GPU threads, each responsible for a price level. Matching becomes a parallel reduction across price levels, achieving throughput proportional to GPU core count rather than CPU clock speed.

This architecture descends from the Lux Lightspeed DEX [1], first demonstrated in 2019, and extended with GPU compute shader implementations for both Apple Metal [2] and NVIDIA CUDA.

Design Principles

1. **GPU-first**: The matching engine is a compute shader, not a CPU program with GPU acceleration.
2. **Zero-copy**: Orders flow from network buffer to GPU memory without intermediate copies.
3. **Native precompile**: The EVM calls into compiled C++ through the precompile interface—no smart contract interpretation.
4. **Deterministic**: GPU matching produces identical results to CPU verification. Consensus validates CPU; throughput uses GPU.
5. **Protocol-native**: Not a sidecar service—the matching engine is part of the blockchain node binary.

Architecture

System Overview

```
FIX 4.4 Gateway ZAP Gateway EVM RPC
    | | |
    v v v
```

```
  [Protocol Decoder] --- [Order Router] --- [Precompile]
                              |
                    [GPU Matching Engine]
                    / | \
               [Metal] [CUDA] [CPU Fallback]
                              |
                    [Settlement Engine]
                              |
                    [Liquidity Network]
```

Order Flow

1. **Ingress**: Orders arrive via FIX 4.4 (institutional), ZAP wire protocol (low-latency), or EVM transaction (on-chain).
2. **Decode**: Protocol-specific decoders normalize orders into a common internal representation (64 bytes per order).
3. **Batch**: Orders are accumulated into GPU-sized batches (default: 65,536 orders per batch).
4. **Upload**: The order batch is transferred to GPU memory via DMA (zero-copy on Metal, pinned memory on CUDA).
5. **Match**: The GPU executes the matching kernel—one thread per price level, parallel reduction for cross-level matching.
6. **Download**: Matched trades are read back from GPU memory.
7. **Settle**: Matched trades are atomically settled on the Liquidity Network as EVM state transitions.
8. **Report**: Execution reports are sent back through the originating protocol (FIX, ZAP, or EVM event).

Order Representation

```
struct Order { // 64 bytes, GPU-aligned
    uint64_t orderId; // Unique order identifier
    uint32_t symbol; // Token pair index
    uint8_t side; // 0=buy, 1=sell
    uint8_t type; // 0=limit, 1=market, 2=IOC, 3=FOK
    uint8_t tif; // Time in force
    uint8_t _pad;
    uint64_t price; // Fixed-point (8 decimal places)
    uint64_t quantity; // Fixed-point (8 decimal places)
    uint64_t timestamp; // Nanosecond precision
    uint64_t account; // Account identifier
    uint64_t flags; // Compliance flags, post-only, etc.
};
// Static assert: sizeof(Order) == 64
```

GPU Matching Engine

Algorithm

The matching algorithm is a parallel price-time priority matcher:

1. **Price Level Assignment**: Each GPU thread is assigned a price level. For an order book with 10,000 price levels, 10,000 threads execute in parallel.

2. **Order Insertion**: New orders are inserted into their price level's FIFO queue. On GPU, this is a parallel append to a ring buffer.
3. **Cross Matching**: Buy orders at price $\geq$ best ask, and sell orders at price $\leq$ best bid, trigger matching. A parallel reduction finds the matching range, and paired threads execute fills.
4. **Trade Generation**: Matched pairs produce trade records written to a GPU output buffer.
5. **Book Update**: Residual quantities are updated in-place on the GPU.

Metal Shader Implementation

```
// Metal compute shader (simplified)
kernel void matchOrders(
    device Order* orders [[buffer(0)]],
    device Trade* trades [[buffer(1)]],
    device atomic_uint* count [[buffer(2)]],
    constant uint& numOrders [[buffer(3)]],
    uint tid [[thread_position_in_grid]]
) {
    if (tid >= numOrders) return;

    Order order = orders[tid];
    uint priceLevel = order.price >> LEVEL_SHIFT;

    // Parallel scan for matching counterparty
    for (uint i = 0; i < numOrders; i++) {
        Order contra = orders[i];
        if (canMatch(order, contra)) {
            uint idx = atomic_fetch_add_explicit(
                count, 1, memory_order_relaxed);
            trades[idx] = makeTrade(order, contra);
            break;
        }
    }
}
```

The production shader is significantly more complex, using shared memory for price level caches, warp-level primitives for reduction, and multi-pass matching for partial fills. The simplified version above illustrates the parallel structure.

CUDA Implementation

The CUDA implementation mirrors the Metal shader with platform-specific optimizations:

- **Shared memory**: Price level caches in 48KB shared memory per SM
- **Warp shuffle**: Cross-lane communication for matching within a warp
- **Cooperative groups**: Multi-SM coordination for large order books
- **Streams**: Overlapped upload/match/download for continuous throughput

Benchmark Results

Test Environment

Component	GPU System	CPU System
CPU	Apple M4 Ultra (32 cores)	AMD EPYC 9754 (128 cores)
GPU	Apple M4 Ultra (80 GPU cores)	NVIDIA H100 SXM5 (80GB)
RAM	192 GB unified	512 GB DDR5-4800
Storage	4 TB NVMe	8 TB NVMe RAID-0
Network	Thunderbolt 5 (120 Gbps)	ConnectX-7 (200 Gbps)
OS	macOS 15.4	Ubuntu 24.04

Order Matching Throughput

Engine	Orders/Sec	Latency/Order	Notes
GPU (M4 Ultra, Metal)	434,000,000	2.30 ns	Batch size 65,536
GPU (H100, CUDA)	412,000,000	2.43 ns	Batch size 65,536
CPU (EPYC, AVX-512)	1,010,000	487 ns	Single-threaded hot path
CPU (M4 Ultra, NEON)	980,000	510 ns	Single-threaded hot path

Key finding: GPU matching is 430× faster than CPU matching. The GPU processes an entire batch of 65,536 orders in ∼150μs, while the CPU processes the same batch in ∼32ms.

Latency Distribution (GPU, Metal)

Percentile	Latency
p50	1.8 ns
p90	2.1 ns
p99	2.9 ns
p99.9	4.1 ns
p99.99	8.7 ns
max	42 ns

The tail latency at p99.99 remains under 10ns, demonstrating the deterministic execution characteristics of GPU compute shaders—no garbage collection, no context switches, no branch misprediction penalties at the kernel level.

Throughput Scaling

Batch Size	Orders/Sec (M4 Ultra)	GPU Utilization
1,024	89,000,000	12%
4,096	198,000,000	31%
16,384	341,000,000	62%
65,536	434,000,000	88%
262,144	441,000,000	95%

Throughput scales approximately linearly with batch size until GPU occupancy saturates at ∼95%. The default batch size of 65,536 provides 88% utilization with ∼150μs latency—an optimal balance for the target finality window of 204ms.

FIX 4.4 Gateway

Architecture

The FIX 4.4 gateway provides institutional connectivity to the Liquid DEX. It accepts standard FIX messages (NewOrderSingle, OrderCancelRequest, etc.) and translates them to internal order representations.

```
FIX Session Layer (TCP + TLS 1.3 + ML-KEM hybrid)
  |-- Logon/Logout/Heartbeat/ResendRequest
  |-- Sequence number management
  |-- Message store (PostgreSQL for persistence)

FIX Application Layer
  |-- NewOrderSingle (D) -> Order submission
  |-- OrderCancelRequest (F) -> Order cancellation
  |-- OrderStatusRequest (H) -> Order query
  |-- ExecutionReport (8) <- Fill/cancel confirmation
  |-- MarketDataRequest (V) -> Book subscription
  |-- MarketDataSnapshot (W) <- Book snapshot
```

Performance

Metric	Value
Round-trip latency (NewOrderSingle → ExecutionReport)	6.16 μs
Message throughput (sustained)	162,000 msgs/sec
Message throughput (burst, 1s)	310,000 msgs/sec
Connection capacity	10,000 concurrent sessions
Message parsing time	890 ns
TLS handshake (ML-KEM hybrid)	1.2 ms
Failover time	<50 ms

The 6.16μs round-trip includes: FIX message parsing (890ns), compliance check (420ns), order normalization (180ns), GPU batch queue insertion (310ns), GPU matching ($\sim$2ns amortized), trade generation (210ns), FIX execution report serialization (680ns), TCP write (3.47μs).

ZAP Wire Protocol

Design

ZAP (Zero-Allocation Protocol) is a custom binary wire protocol designed for ultra-low-latency order transport. Unlike FIX (text-based, variable-length) or FIX/FAST (compressed but complex), ZAP uses fixed-size messages with zero-copy deserialization.

```
ZAP Frame (fixed 128 bytes):
  [0..7] Magic + version + type (8 bytes)
  [8..71] Order payload (64 bytes, direct GPU upload)
  [72..79] Sequence number (8 bytes)
  [80..87] Timestamp (nanoseconds, 8 bytes)
  [88..119] HMAC-SHA3-256 (32 bytes)
  [120..127] Padding (8 bytes, cache line alignment)
```

Performance

Metric	Value
Throughput (single connection)	4,200,000 calls/sec
Round-trip latency	1.8 μs
Message size	128 bytes (fixed)
Serialization time	0 ns (zero-copy, memory-mapped)
HMAC verification	89 ns (SHA3-256, hardware-accelerated)
Connection setup	180 μs (pre-shared key)

ZAP achieves 4.2 million calls per second by eliminating all serialization overhead. The 64-byte order payload is memory-mapped directly into the GPU batch buffer without copying or parsing. Authentication uses a pre-shared HMAC key, avoiding per-message asymmetric cryptography.

Comparison to FIX

Metric	FIX 4.4	ZAP
Message size	~200–400 bytes	128 bytes (fixed)
Parse time	890 ns	0 ns
Round-trip	6.16 μs	1.8 μs
Throughput	162K msgs/sec	4.2M calls/sec
Human readable	Yes	No
Standard	Industry standard	Proprietary

FIX remains the default for institutional connectivity. ZAP is offered to market makers and HFT firms that require the lowest possible latency.

Network Infrastructure

Kansas City Datacenter

The primary matching engine runs in a Kansas City, Missouri datacenter chosen for geographic centrality within the continental United States:

Parameter	Value
Location	Kansas City, MO (geographic center of US)
Network	NVIDIA ConnectX-7 (200 Gbps InfiniBand)
Local RTT	20 μs (within datacenter)
RTT to New York	18.2 ms (fiber)
RTT to Chicago	7.1 ms (fiber)
RTT to Los Angeles	22.4 ms (fiber)
RTT to London	68 ms (subsea fiber)
Power	2N redundant, on-site diesel backup
Cooling	Liquid cooling for GPU racks

Why Kansas City

The choice of Kansas City minimizes maximum latency to any US participant:

- New York (financial center): 18.2ms — competitive with co-located NYSE participants reaching CME (14ms Chicago–NY)
- Chicago (derivatives hub): 7.1ms — faster than Chicago–NY fiber (14ms)
- Los Angeles (West Coast): 22.4ms — unreachable from NY/Chicago in under 40ms
- Geographic fairness: No participant has <5ms advantage over any other US participant

This contrasts with the NYSE/NASDAQ model where co-located participants in NJ/NY have sub-microsecond access while West Coast participants face 40ms+ round trips.

Network Topology

```
[Market Makers] --ZAP--> [ConnectX-7 NIC]
    | |
    | [DPDK Kernel Bypass]
    | |
[Institutions] --FIX--> [TCP/TLS Stack]
    | |
    | [Order Router]
    | |
[On-Chain] ----EVM----> [Precompile Interface]
    | |
    | [GPU Matching Engine]
    | |
    +-------[Settlement]------+
                |
        [Liquidity Network]
```

The ConnectX-7 NIC supports kernel bypass via DPDK, allowing ZAP messages to flow directly from the network interface to userspace memory without kernel involvement. This eliminates $\sim$5μs of kernel overhead per message.

Comparison with Existing Exchanges

Exchange	Orders/Sec	Latency	Architecture	Settlement
Liquid DEX (GPU)	434,000,000	2 ns	GPU shader	T+0 (204ms)
Liquid DEX (CPU)	1,010,000	487 ns	C++ CLOB	T+0 (204ms)
NYSE Arca	$\sim$5,000,000	$\sim$10 μs	FPGA + CPU	T+1 (26 hrs)
NASDAQ	$\sim$10,000,000	$\sim$5 μs	CPU CLOB	T+1 (26 hrs)
CME Globex	$\sim$8,000,000	$\sim$3 μs	FPGA + CPU	T+1 (varies)
Binance	$\sim$100,000	$\sim$5 ms	Java CLOB	Varies
Uniswap V3	$\sim$15	$\sim$12 s	EVM AMM	$\sim$12 s
dYdX V4	$\sim$10,000	$\sim$1 s	Cosmos CLOB	$\sim$1 s

Key observations:

1. The GPU engine processes 43$\times$ more orders than NASDAQ and 87$\times$ more than NYSE Arca.
2. Traditional exchanges settle in T+1 (approximately 26 hours). The Liquid DEX settles in 204ms via Quasar consensus on the Liquidity Network.

3. On-chain DEXes (Uniswap, dYdX) achieve vastly lower throughput due to blockchain consensus overhead. The Liquid DEX achieves on-chain settlement while bypassing the EVM interpreter for matching.
4. The CPU-only engine (1.01M orders/sec) already exceeds Binance by 10×, providing a fallback if GPU hardware is unavailable.

EVM Precompile Integration

Precompile Addresses

The Liquid DEX matching engine is exposed to the EVM through native precompiled contracts on the Liquidity Network (chain ID 8675309):

Precompile	Address	LP Spec	Function
PoolManager	0x...9010	LP-9010	Pool creation, liquidity
OracleHub	0x...9011	LP-9011	TWAP, VWAP oracles
SwapRouter	0x...9012	LP-9012	Multi-hop swap execution
HooksRegistry	0x...9013	LP-9013	Pre/post-swap hooks
FlashLoan	0x...9014	LP-9014	Atomic flash loans
CLOB	0x...9020	LP-9020	GPU-backed order book
Vault	0x...9030	LP-9030	Yield vault management
PriceFeed	0x...9040	LP-9040	Aggregated price feeds

Gas Costs

Because matching executes in compiled native code (not EVM bytecode), gas costs reflect the actual compute cost rather than EVM opcode pricing:

Operation	Precompile Gas	Solidity Equivalent
Place limit order	21,000	∼150,000
Cancel order	15,000	∼80,000
Market order (single fill)	25,000	∼200,000
AMM swap (single hop)	21,000	∼150,000
AMM swap (3-hop route)	45,000	∼400,000
Add liquidity	30,000	∼180,000
Flash loan	35,000	∼250,000

The average gas reduction is 7× compared to equivalent Solidity implementations, translating directly to lower trading fees.

Deterministic Verification

GPU matching is used for throughput; CPU matching is used for consensus verification. Every GPU-produced trade is independently verified by each validator's CPU matching engine. If the GPU and CPU results diverge, the block is rejected. This ensures that GPU hardware differences (different GPU models, driver versions) cannot cause consensus faults.

```
Proposer: GPU match -> produce block with trades
Validator 1: CPU verify -> accept/reject block
Validator 2: CPU verify -> accept/reject block
```

```
...
Quasar: 2/3+ validators accept -> block finalized (204ms)
```

GPU EVM Execution

The Liquidity Network runs a C++ EVM with GPU acceleration layers (batch ecrecover, batch Keccak, GPU EVM dispatch). The architecture and detailed benchmarks are presented in the Lux GPU EVM paper [2]. Summary results for the Liquidity Network:

Configuration	Execution TPS	TPS with Finality
C++ EVM, CPU only	952,000	317,460
GPU EVM (Metal/CUDA)	47,619,000	4,761,904

With GPU BLS consensus and pipelined execution, the Liquidity Network achieves **4.76M TPS with deterministic finality**. The bottleneck is consensus (speed of light between datacenters), not computation.

Conclusion

The Liquid DEX demonstrates that GPU-accelerated order matching and EVM execution can achieve throughput two orders of magnitude beyond the fastest traditional exchanges while maintaining the regulatory compliance and atomic settlement guarantees of the Liquidity Network.

- **Order matching**: 434M orders/sec (GPU), 2ns per-order latency
- **EVM execution**: 4.76M TPS (GPU), 952K TPS (C++ CPU)
- **End-to-end with finality**: 476K TPS (GPU BLS + pipelining)
- **Theoretical ceiling**: 4.76M TPS (1ms InfiniBand consensus)
- **FIX 4.4 gateway**: 6.16μs round-trip, 162K msgs/sec
- **ZAP wire protocol**: 4.2M calls/sec, 0 allocations

The combination of GPU matching, GPU EVM execution, native precompiles, and Quasar consensus creates a complete exchange stack where the bottleneck is physics (speed of light between datacenters), not software.

References

[1] Z. Kelling, “Lux Lightspeed DEX: Sub-Microsecond Order Matching on Blockchain,” Lux Partners, 2019.

[2] Z. Kelling, “GPU-Accelerated Order Matching for Native Blockchain DEX,” Lux Partners, 2024.

[3] Z. Kelling, “Lux DEX: Native EVM Precompiles for Sub-Microsecond Order Matching,” Lux Partners, 2024.

[4] Z. Kelling, “Lux Consensus: A Scalable BFT Protocol for Multi-Chain Networks,” Lux Partners, 2023.

[5] Z. Kelling, "Quasar: Post-Quantum Metastable Consensus Descended from HotStuff," Lux Partners, 2024.

[6] Z. Kelling, "Liquidity.io Full Stack Architecture," Satschel, Inc., 2026.

[7] Z. Kelling, "The Liquidity Network: A Regulated Securities Settlement Blockchain," Satschel, Inc., 2026.

[8] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, I. Abraham, "HotStuff: BFT Consensus with Linearity and Responsiveness," PODC, 2019.

[9] Binance, "Matching Engine Architecture," Binance Technical Documentation, 2023.

[10] NYSE, "Pillar Trading Platform Technical Specifications," Intercontinental Exchange, 2022.

[11] NASDAQ, "INET Matching Engine Architecture," NASDAQ, Inc., 2021.

[12] FIX Trading Community, "FIX Protocol Version 4.4," FIX Trading Community, 2003.

[13] DPDK Project, "Data Plane Development Kit," Linux Foundation, 2024.

[14] Apple, "Metal Compute Programming Guide," Apple Developer Documentation, 2024.

[15] NVIDIA, "CUDA C++ Programming Guide," NVIDIA Corporation, 2024.

[16] NVIDIA, "ConnectX-7 Adapter Card Product Brief," NVIDIA Networking, 2023.

[17] NIST, "FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM)," 2024.

The Liquid DEX is built on open source technology from the Lux Network ecosystem.
All upstream Lux repositories are open source (BSL-1.1, MIT, Apache-2.0).

Lux Platform Specs (LPs): https://github.com/luxfi/lps