

Article

An Enhanced Latency-Bounded GPU-Resident Pipeline for Real-Time Market Stream Visualization

Donia Y. Badawood ¹  and Fahd M. Aldosari ^{2,*} ¹ Department of Data Science, Umm Al-Qura University, Makkah 21955, Saudi Arabia; dybadawood@uqu.edu.sa² Department of Computer and Networks Engineering, Umm Al-Qura University, Makkah 21955, Saudi Arabia

* Correspondence: fmdosari@uqu.edu.sa

Abstract

High-Frequency Trading (HFT) dashboards require rapid reception, aggregation, and visualization of order book and trade update streams that may arrive at multi-million message rates. Conventional CPU-based and CPU-GPU hybrid visualization pipelines can suffer from significant delays during periods of burst due to CPU-mediated rendering, synchronization, kernel launch overhead, and copies on the host. This paper presents a visualization pipeline that is entirely resident on the graphics processor with zero-copy access to NIC accessible pinned buffers, persistent CUDA processing, fused stage execution of the parse-aggregate pipeline, and persistent CUDA OpenGL buffer interoperability. The goal is not to reach production status but rather to see whether host-to-host data movement can be decreased and whether the stages of GPU processing can be consolidated to improve latency, throughput and frame cadence in controlled HFT-style workloads. The evaluated workstation achieved a mean ingest-to-pixel latency of 6.3 ms using the proposed design compared to 29.4 ms for the current design, with sustained throughput of 10.2 million messages per second, which is 20 times greater than the current design, and a steady-state range of 185 to 192 frames per second with a burst floor of 178 frames per second for the proposed design. The improvement observed can be attributed to both the zero-copy ingestion and fused persistent kernel execution. Based on the obtained results, the proposed method of use of this technique in the implementation of real-time financial visualization under the proposed conditions is possible. More general testing is still required on other NICs, other generations of GPUs and PCIe configurations, workload traces, and actual exchange feeds.

Keywords: high-frequency trading (HFT); real-time data visualization; GPU acceleration; low-latency processing; algorithmic trading; financial market data; data throughput optimization; zero copy network



Academic Editor: Filippo Palombi

Received: 15 May 2026

Revised: 8 June 2026

Accepted: 13 June 2026

Published: 17 June 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

High-frequency trading (HFT) needs an order flow reaction rate of sub-milliseconds, and practical real-time systems should be able to not only display market conditions but also calculate fast enough to enable both human and automated strategies to respond. As a matter of fact, dashboards, depth views, and anomaly monitors can only be as useful as their end-to-end latency permits. The necessity to fulfill this condition is motivated by the existing literature and practice of previous finance, and its impact on performance and market efficiency [1,2]. Conventional CPU-based visualization stacks are unable to scale to

HFT rates due to the serialization of rendering and state update of cores and memory copies across cores and memory copies [3,4]. Consequently, visual feedback falls behind the feed in bursts to diminish the quality of decisions and situational awareness. These limitations are reflected in earlier research on interactive data displays and real-time analytics in HFT, which encourages additional parallel visual pipelines [5–7]. The GPUs provide enormous data-parallel performance and are now considered standard in scientific work and finance and can be used to perform quick simulations, ML inference, and rendering. Modern GPUs provide the parallelism and memory bandwidth required for real-time visualization and processing of market data, although they were initially developed for graphics-oriented tasks [8,9]. The reliability and operational risk aspects that are paramount in HFT production environments further increase the demand for pipelines that reduce latency and jitter under stress [10]. But in itself, being able to use a GPU is not enough. Traditional host-device pipelines may have the overhead of copying and launching/synchronization costs, which can significantly decrease the theoretical advantage.

The development of HFT and the volume of data it generates increase the importance of real-time visualization as a component of the system as they support strategy-making, monitoring, and risk management when markets produce millions of updates per second. The importance of scalable, low-latency financial visual analytics [1,5,11,12] has been highlighted by earlier studies and systems and is encouraged by the general efficiency of GPU computing. This study's methodological gap is the lack of an experimentally tested visualization pipeline that considers packet ingestion, updating the state of the order book, generation of visualization primitives, and the rendering synchronizations as one single ingest to pixel path. Most resident streaming systems on GPUs optimize compute kernels or stages of data analytics, while most real-time visualization systems optimize rendering or interaction after data has been staged by the host. FPGA-based HFT systems offer lower network side latency but are less flexible when it comes to graphics interoperation and commodity GPU rendering. Selected operations can be accelerated using hybrid CPU-GPU systems, but these still maintain host-mediated copies, different launch stages or CPU controlled synchronization. The proposed framework pushes these methods further and includes fused persistent CUDA processing and persistent CUDA/OpenGL interoperation in one latency aware visualization loop that also enables zero copy packet access. This study is based on the following research questions.

- RQ1: How far can GPU acceleration improve the speed and efficiency of real-time HFT visualization relative to CPU and hybrid baselines?
- RQ2: Which optimization techniques—zero-copy memory access, kernel fusion, batching, buffer depth—most reduce end-to-end latency without harming stability?
- RQ3: How do these designs behave under bursty/volatile workloads typical of live markets in terms of p99/p99.9 latency, drop rate, and FPS stability?

This paper presents a pipeline designed to display the HFTs in a device-resident manner and the results of tests with a controlled workstation testbed. The tested workload yielded a mean ingest-to-pixel latency of 6.3 ms, sustained throughput of 10.2 MPS and ~190 FPS render with a steady-state range of 185 to 192 FPS and a burst floor of 178 FPS. These are provided as statements of testbed values, rather than general guarantees. The evaluation aims to separate the zero-copy ingestion performance, kernel-fused processing performance, persistent execution performance and CUDA OpenGL interoperation performance under the same baseline conditions. The main contributions of this study are as follows.

- (1) A GPU-resident ingest-to-pixel pipeline is proposed for HFT-style market stream visualization. The pipeline minimizes host-mediated data movement by mapping NIC

accessible pinned buffers into the CUDA address space and by keeping visualization primitives in persistently mapped CUDA OpenGL buffers.

- (2) A fused persistent CUDA processing operator is designed to parse incoming records, update the order book state, and stage visualization vertices in a single pass. This design reduces kernel launch overhead, intermediate memory sweeps, and synchronization points compared with multi-kernel GPU processing.
- (3) A latency-bounded rendering policy is introduced to maintain visual responsiveness during burst periods. The policy permits intermediate frame skipping when the frame budget is exceeded, while preserving the processing of all order book update packets and tracking packet loss separately from visual frame drops.
- (4) A controlled evaluation is reported using CPU, CPU-GPU hybrid, multi-kernel GPU, zero-copy-only GPU, and fused zero-copy GPU variants under identical workloads. The evaluation reports ingest-to-pixel latency, throughput in MPS, frame rate stability, energy-normalized throughput, and ablation results.

The remainder of the paper is organized as follows: Section 2 presents a literature review of HFT systems, GPU-based visualization, and low-latency pipelines. Section 3 describes our architecture, the GPU processing path, and the implementation. Section 4 presents a set of results, including latency, throughput, FPS, and energy, along with ablations and stress tests. In Section 5, limits and future directions, including more comprehensive coverage of the market and deployment, are discussed.

2. Related Work

HFT real-time visualization is at the boundary between ultra-low-latency market engines, interactive visual analytics, and high-performance visual computing. The previous literature determines the sensitivity of electronic markets to latency and promotes engineering stacks to reduce end-to-end time between feed ingress and pixels on a screen [13,14]. The description of the impact of microseconds of latency on the quality of execution and the viability of strategies to co-location, kernel-bypass network, and hardware-proximate processing in production stacks inspired seminal market-microstructure and systems work [2,15]. Besides software tuning, FPGA-based systems exhibit deterministic, NIC-close processing routes, which lower jitter, one of the applications where software copies and host mediation can be minimized in a practical manner [11]. Domain-specific visualization systems (e.g., depth view and event-stream explorers) focus more on task-oriented encodings and interactions on limit-order books. Still, they are typically based on CPU-oriented rendering or offline processing [5]. Wider real-time analytics models monitor streaming dashboards and metric computation but not aircraft rendering or data movement and do not quantify ingest-to-pixel performance at bursty loads or ingest performance at bursty loads [6,16]. Scalable display architectures are aimed at multi-display rendering instead of wire-speed NIC-to-GPU paths common in HFT [17]. Lockwood et al. [18] demonstrates that host-stack latency in HFT can be reduced by FPGA-resident parsing. Yet, the control/visualization path and pipeline are still CPU-centric, with no exposure of NIC buffers to the GPU address space. The current work aims to achieve the same latency with a fully software-resident pipeline consisting of zero-copy ingest. Singh et al. [11] Surveys autonomous visualization in real-time and suggests that accelerators are necessary to service interactive latencies but provides no information on ingest buffers or render interop. This paper adds a realistic GPU implementation that is pageable and staging-free, and it also measures end-to-end benefits.

In Petrosanu and Pirjan [19], throughput-per-dollar is analyzed as well as energy efficiency but without considering packetized market data or frame pacing. Those efficiency assertions are operationalized in a latency-sensitive environment and reported to be 34,000 MPS/W. Cao et al. [20] optimized HFT strategies using deep reinforcement learning,

but their work mainly assumes a host-centric I/O setting. In contrast, the proposed device-resident pipeline extends this direction by reducing ingest-to-visualization latency and stabilizing the update cadence. Prior journal work on low-latency market data feed arbitration has shown that network-level acceleration, protocol-aware feed handling, and reduced software-side processing are important for time-critical financial data pipelines [21]. These design principles motivate the proposed SoA-based order book representation and indexed receive ring design. However, existing low-latency HFT architectures do not directly address NIC-to-GPU zero-copy access, fused persistent CUDA processing, or persistent CUDA OpenGL interoperation, which are the main concerns of this work. Scientific and financial workloads increasingly use GPUs for parallel simulation, inference, and accelerator native processing, while recent finance-focused studies mainly emphasize training and inference pipelines rather than complete ingest-to-pixel visualization paths [22–24]. Recent studies using accelerators have demonstrated GPUs’ ability to increase throughput. Yet, frequent transfers of data from the CPU to the GPU are a significant limitation for streaming query execution [25]. Likewise, recent visualization work that leverages real-time CUDA has pointed out that traditional approaches that involve repeated GPU-to-CPU copies limit interactivity and that the process of graphics interoperation should be well managed to maintain efficient computation and rendering [26]. For low-latency financial applications, recent studies also pointed out the need for cache-aware programming, lock-free data transfer, and statistically sound benchmarking of workloads similar to HFT [27]. Table 1 presents a data-enriched comparison of prior work. Unlike previous CPU-based visualization stacks [1,5,11,12], the design is entirely based on the graphics card, and there are no host-side copies of data as an ingestion with zero-copy memory usage is made, and interoperation with other systems is maintained. Hybrid pipelines [18] unload aggregation or rendering partially but still incur transfer overheads and launch synchronization costs. This pipeline combines parsing, aggregation, and rendering in a single pass running on a single device, with a latency of less than 10 ms A-to-P and stable 190 FPS frame rates. In comparison with the deterministic systems of the FPGA type [11], the given model is able to reach similar latency with flexibility and portability across commodity GPUs. Explicitly scaled energy-normalized throughput (34,000 MPS/W) also realizes prior theoretical efficiency assertions [19], showing that visually performant applications that use GPUs can achieve HFT rates.

Table 1 shows that prior research addresses important parts of low-latency financial computing but not the complete visualization path studied here. Market microstructure and HFT systems research focuses on the reasons behind latency, but typically it is measured in terms of execution or trade latency, not freshness of the visual state. FPGA-based systems minimize the jitter near the network interface but do not directly deal with commodity GPU rendering or CUDA/OpenGL interoperation. Interactive visualization systems offer good order book views, typically need to assume that the data resides on the host system and do not report on timing at the burst level between the NIC and the display. The high device throughput of GPU computing studies is demonstrated, though most only do isolated kernels, not persistently ingest to render loops.

Table 1. Data-enriched comparison of prior work.

Ref.	Domain/Focus	Pipeline Type	Data Movement Path	Visualization Focus	Reported Latency Metric	Throughput/FPS Reported	Evaluation Data	Noted Limitation vs. This Work
[1]	HFT systems and practice	—	—	Not visualization-centric	Qualitative latency and design considerations	—	Real-market practice	No ingest → render measurements; no GPU pipeline detail

Table 1. *Cont.*

Ref.	Domain/Focus	Pipeline Type	Data Movement Path	Visualization Focus	Reported Latency Metric	Throughput/FPS Reported	Evaluation Data	Noted Limitation vs. This Work
[2]	Low-latency trading microstructure	—	—	Not visualization-centric	Empirical latency effects on trading	—	Market data studies	Focus on execution latency; not end-to-end visualization
[11]	FPGA library for HFT	FPGA	NIC → FPGA (hardware)	Not visualization	Deterministic pipeline timing	—	Hardware prototypes	Hardware-centric; no NIC → GPU or rendering path
[5]	Visual exploration of HFT data	CPU-centric	Host-resident	Depth/trade views	—	—	Curated datasets	Not engineered for wire-speed ingest → pixel latency
[12]	Large tiled displays	Distributed CPU/GPU	Networked display	Multi-display rendering	—	—	System deployments	Scales displays; not HFT-rate NIC → GPU under burst conditions
[6,7]	Streaming analytics	CPU-first	Host copies	Dashboards/metrics	—	—	Case studies	Treat data motion abstractly; no GPU interop or tail-latency analysis
[8,9]	HPC/training/inference	GPU compute	Device-resident compute	Model pipelines	Kernel/runtime metrics	—	Benchmarks	Compute-centric; not ingest → render visualization

(Dashes indicate metrics not reported.).

3. Methodology

The HFT visualization must respond within the millisecond range when processing multi-million-message workloads and microbursts caused by volatility. The system aims at achieving three main goals, namely: (i) reduce end-to-end ingest-pixel latency, (ii) maintain wire-rate throughput with constant frame cadence, and (iii) maintain bounded tail latency at load conditions whilst maintaining high energy efficiency. A discrete GPU can be found in a single workstation that also has a discrete display. Market data is provided as a high-rate, unidirectional UDP stream (limit order book and trade updates). The NIC writes packets to pinned host memory, and CUDA maps them into the device address space, where they can be accessed by GPU kernels with zero-copy. A second optional GPUDirect RDMA path is used to access GPU memory when supported. The CPU performs only queue housekeeping (posting receive and advancing the producer index), and the path between ingestion and rendering is within the device.

3.1. System Architecture and Design

The architecture puts a device-resident pipeline in place, which transfers market data off the network interface to the display; the fast path does not involve host-side copies. Data ingress, GPU processing, render interoperation, and scan-out are intended to make certain that packet bytes, state updates, and visualization vertices stay on the device. Meanwhile, the CPU offers very minimal coordination. The desired results of this organization include low median latency and tails bounded at bursts and a fixed frame cadence of 190 FPS.

The NIC inserts packets in a receive (RX) depth D ring. RX buffers are pinned in the host memory and mapped into the global address space of the GPU through CUDA zero-copy, where the device pointers are made available to the host pages. RX descriptors have (addr, len, seq); producer (head) and consumer (tail) indices, which are stored in pinned memory to allow coordination between NIC/CPU and GPU with a low overhead. A non-compulsory NIC-to-GPU path based on GPUDirect RDMA shunts descriptors of RXes to buffers in the GPU in case of their assistance. Figure 1 represents the organization as a whole.

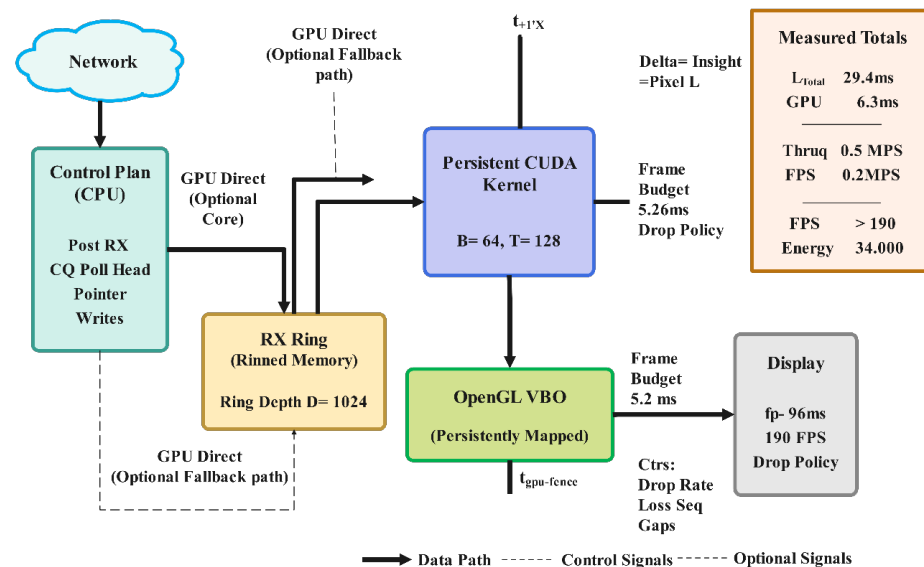


Figure 1. End-to-end architecture of the proposed GPU resident market stream visualization pipeline. The diagram shows the network ingress path, pinned RX ring, persistent CUDA kernel, CUDA OpenGL vertex buffer interoperation, frame pacing policy, and display output under the evaluated HFT style UDP workload.

GPU processing is performed by a persistent CUDA kernel that polls the RX ring and consumes newly completed descriptors in batches of size B . Packet payloads are read in place through zero-copy device pointers, parsed into protocol records, and applied to the order book state stored in a structure-of-arrays layout ($price[]$, $qty[]$, $side[]$, $ts[]$) to maximize coalesced access. Parsing, aggregation, and vertex staging are fused into a single pass to eliminate intermediate device buffers and kernel-launch overheads; warp-aggregated atomics mitigate contention on hot price levels, and a tiling policy with tile size T enforces regular memory access.

Render interoperation keeps visualization data on the device. Visualization primitives (depth bars, trend lines, and candlesticks) are written directly into an OpenGL vertex buffer object that remains persistently mapped into CUDA; a GPU fence issued at the frame boundary orders between compute and draw without per-frame map/unmap or CPU mediation. V-sync is disabled to avoid scan-out coupling. The frame-time budget for the 190 FPS target is 5.26 ms; when instantaneous workloads exceed this budget, a latency-bounded policy drops intermediate frames rather than allowing queue growth, thereby controlling tail latency while preserving ingest correctness and book state.

The CPU executes a minimal control plane: posting receives, advancing the head index, and polling completions. No host-side memory copies occur on the fast path. RX sequence numbers provide on-device loss detection; packet-loss counters are maintained separately from visualization frame-drop counters to preserve operational clarity.

3.2. System Goals and Model

There are three goals that guide the design. First, the pipeline should minimize host-mediated data transfer since explicit host-to-device staging can be a significant contributor to latency for small messages and a high frequency of message access. Second, the update and rendering path should avoid too many synchronization points as different parsing/aggregation and vertex staging launches will add repeated scheduling and memory traversal overhead. Third, during the burst period, the visual responsiveness should not be lost by dropping out-of-order book updates. The use of zero-copy packet access, a fused persistent CUDA kernel, and persistent CUDA OpenGL interoperation is motivated

by these goals. The proposed design will thus be a streaming systems problem, where the optimization for data movement, computation, and rendering synchronization should be optimized together, rather than independently.

3.3. Ingestion and Zero-Copy Access

The ingestion stage aims to minimize the number of data movements through the network interface to the GPU processing path. Updates from the market arrive via a high-rate UDP stream and are added to a receive ring which is stored in pinned host memory. The packet address, length, and sequence number ($addr, len, seq$) are included in each receive descriptor. A persistent GPU kernel can directly access packet payloads via device visible pointer as the pinned buffers are mapped into CUDA address space. This design eliminates host-to-device copies that must be performed explicitly on the fast path, and minimizes the staging overhead typically incurred when initial packets are first copied into intermediate CPU buffers, before being staged to GPU memory.

The receive ring is indicated as $RX[D]$, where D is the ring depth. The producer index $Head$ is advanced by the CPU or NIC completion path after new packets arrive, whereas the consumer index $Tail$ is maintained by the GPU kernel after packets are consumed. The available number of kernel iterations is calculated for each kernel as follows:

$$A = (Head - Tail + D) \bmod D \quad (1)$$

where A denotes the number of descriptors currently available for GPU processing. The kernel processes at most B packets per iteration, where B is the maximum batch size. Therefore, the effective batch size is computed as

$$b = \min(A, B) \quad (2)$$

By processing packets in a batched fashion, the GPU can process packets in a predictable manner and limit unnecessary synchronization between the CPU and GPU. The $Head$ and $Tail$ indices are inserted in a release/acquire fashion, and a system-level memory fence is written out after the GPU moves $Tail$. This allows for packet consumption to be reported to the CPU and NIC control path, without the additional host-side copies.

The primary benefit of zero copy access is to significantly decrease the explicit staging term in the latency path. In a copy-based implementation, a block of S bytes needs to be copied from host to device prior to GPU processing. The staging overhead is about

$$L_{copy} = \frac{S}{BW_{PCIe}} + L_{launch} \quad (3)$$

where BW_{PCIe} is the effective PCIe bandwidth, and L_{launch} is the launch/scheduling overhead for each separate processing stage. The proposed zero copy path is that the packet payload is read directly from a mapped pinned buffer by the GPU kernel. This does not eliminate the rest of the interconnect cost because mapped host memory is still accessed down the PCIe path. But it eliminates another explicit copy and speeds up small and often-market update messages.

To prevent visualization frame drops from being confused with packet loss, the packet sequence numbers are utilized. On packet consumption, the GPU holds up the packet if the sequence number it saw does not match the sequence number it expects. If a gap is found, then a device-resident packet loss counter is incremented. Unlike the frame drop statistics, this counter is not reported along with the frame drop since a skipped visual frame does not necessarily mean that the order book updates are not lost. The proposed

pipeline still sends all book update packets received even though the renderer does not perform intermediate rendering steps during a burst.

The frame pacing policy is triggered if the predicted rendering work is greater than the frame interval or if the queue's length reaches a configured limit. The decision rule is given by

$$\text{drop_frame} = \begin{cases} 1, & \text{if } t_{pred} > t_{frame} \text{ or } Q_{occ} > Q_{thr} \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

where t_{pred} is the predicted rendering work, $t_{frame} = 5.263$ ms for the 190 FPS target, Q_{occ} is the receive queue occupancy, and Q_{thr} is the configured queue threshold. When $\text{drop_frame} = 1$, the renderer skips an intermediate visual refresh, but the GPU continues processing market update packets and updating the order book state. Thus, the pacing policy is intended to preserve visual responsiveness and prevent queue growth without discarding market data.

3.4. GPU Data Structures and Kernel-Fused Processing

The GPU uses device memory to store all the commonly used state and then runs a single persistent kernel that processes packets, resets order book structures, and stages visualization primitives in a single pass. This organization does not have intermediate device buffers as well as redundant kernel-launch overheads, and all the hot data remains on the GPU during the ingest→render loop.

(1) Data layout and memory residency

The state of order books is presented in the form of a structure-of-arrays (SoA) to ensure that coalesced memory access is maximized. Each instrument has four parallel arrays in global memory: *price* [] (32-bit integer ticks), *qty* [] (32-bit integer sizes), *side* [] (8-bit sign or 0/1 flag), and *ts* []. Arrays are 128-byte-aligned and zero-padded to the closest multiple of 32 elements in order to have a warp read and write sequential software-defined cache lines. Repeated access is used to store per-instrument metadata (current best bid/ask index, depth limits, and sequence cursors) in a compact header and cache in L1/L2.

In direct read, incoming packet descriptors are read using zero-copy device pointers pointing to pinned host pages. Descriptors store (*addr*, *len*, *seq*). A consumer index, Tail device-resident, increases monotonically with batch processing. The vertex buffer object (VBO) of OpenGL is continuously allocated to CUDA space: there is a 32-bit device counter *vtx_head* , which can be used to offer an atomic reservation to newly generated vertices. VBO capacity V_{max} is made large enough such that a frame worst-case production does not overrun (empirically made larger than the 99.9-percentile demand in stress tests).

(2) Fused parse–aggregate–stage pass

Every time the kernel fetch puts up to $B = 64$ new packet descriptor completions on the RX ring and performs them in $T = 128$ record tiles to enforce regular access to the memory. Warp granularity is the warp-wise cooperation of threads. We load field contents in protocols using fixed-offset loads and use variable-length loads computed using predicated pointer arithmetic to avoid divergence. The atoms of the SoA arrays are updated: these threads of the warp that touch the same price level issue their deltas first using a warp-aggregated atomic `__reduce_add_sync`, and only one lane of the warp issues an atomicAdd to global memory, reducing contention at high levels along the touch-line. Trend line, candlestick, and depth bar vertices are in staged execution whenever the state is updated. Every thread takes a continuous slice in the VBO by `atomicAdd(&vtx_head, k)`, in which k is the number of emitted vertices, and the coalesced stores are subsequently written with the vertex coordinate and attribute values. Reaching a frame boundary, the

kernel gives a device flag that indicates that a lightweight host call or some other step of the GPU can be used as a fence by the renderer.

(3) Correctness, ordering, and loss tracking

Packet sequence numbers seq are used to provide application idempotency and to reveal gaps. Doing the same with the next expected value of the stream, the kernel makes the next step, updates a device counter on gaps detection, and proceeds to process further packets. Visualization can discard frames to fit the frame budget of 5.26 ms; however, book updates will never be discarded. The ingest/render memory ordering is set up with a device-side release semantic prior to the fence; the renderer sees a consistent VBO when the fence is signaled.

(4) Computational cost

Let n represent the number of records read in a batch and m the number of individual levels of price that were touched. The fused pass calculates $O(n)$ parsing and $O(m)$ updates, and each warp updates a single global atomic at a different level since the levels are aggregated; the number of J warps per warp of a typical order book traffic is $m \ll nm$. Each packet tile is read once sequentially to memory traffic, each slot touched is read-modify-written once, and a single sequential write is done to the VBO. The pass-fused compares to the unfused pass in that it avoids two global memory sweeps and two launches, equivalent to the empirical 29.4 ms to 6.3 ms drop in end-to-end latency, and permits sustained 10.2 MPS and 190 FPS. Table 2 provides a summary of the symbols in Algorithm 1.

Table 2. Notation summary for fused parse–aggregate–stage persistent kernel.

Symbol/Variable	Description
D	Ring depth (number of RX descriptors, e.g., 1024)
B	Batch size (packets processed per kernel iteration, e.g., 64)
T	Tile size for warp-coalesced processing (e.g., 128)
RX[D]	Array of GPU-visible packet descriptors {addr, len, seq}
Head, Tail	Producer/consumer indices in pinned memory
Book	Structure-of-Arrays {price[], qty[], side[], ts[]} representing order book state
VBO	Persistently mapped OpenGL vertex buffer in device memory
vtx_head	Atomic cursor within the VBO tracking the next vertex slot
frame_boundary()	Function returning true when a new frame should be rendered
signal_gpu_fence()	GPU synchronization fence to trigger rendering

Algorithm 1 presents the fused parse–aggregate–stage (persistent kernel, $B = 64$, $T = 128$).

This kernel represents the key design decisions of the methodology: direct utilization of network payloads by placing them on zero-copy device pointers, a one-pass combination of parsing and aggregation, vertex staging, and device interoperation. The resulting memory pattern will still be regular at the warp granularity, aggregate contention will be smaller due to aggregation, and the amortization of end-to-end overheads will be through persistence-together, delivering the measured amortization of end-to-end improvements.

3.5. Rendering Interop and Frame Pacing

The 190 FPS goal is set at a 5.263 ms frame interval. A strict upper bound is not assumed for each packet-level ingest-to-pixel latency within this interval. Rather, it specifies the frequency of the rendering. The 6.3 ms reported latency is the average time from packet reception to the time until the GPU fence before draw is visible. Ingestion, GPU state update, staging vertex data, and display cadence are all working on an overlapped pipeline, so even with a mean ingest-to-pixel latency in the ballpark of one frame interval, the

renderer is still at ~190 FPS. For clarity, we report ingest-to-pixel latency and frame cadence separately in this study. If the rendering work that is predicted is greater than the 5.263 ms frame interval, the pacing policy will not render intermediate visual frames but will still apply order book update packets. Thus, the method should be understood not to imply that each and every packet will be seen within a single 5.263 ms frame interval but rather that frame cadence will be maintained under the workload tested.

Algorithm 1. Fused persistent GPU kernel for parse aggregate stage processing

Input:

RX[D]	GPU visible RX descriptors containing address, length, and sequence number
Head	Producer index updated by the CPU or NIC completion path
Tail	Consumer index maintained by the GPU
Book	Device resident order book arrays
VBO	Persistently mapped OpenGL vertex buffer
B	Maximum batch size
T	Tile size

Output:

Updated order book state
 Updated visualization vertices
 Packet loss counter
 Frame ready flag

```

1.  while application is running do
2.      available = (Head – Tail + D) mod D
3.      batch = min(available, B)
4.      if batch = 0 then
5.          continue
6.      end if
7.      for offset = 0 to batch step T do
8.          parallel for each descriptor in current tile do
9.              packet = read packet payload through zero copy device pointer
10.             record = parse packet fields
11.             if record.sequence_number is not expected then
12.                 increment packet_loss_counter
13.             end if
14.             level = map record price to order book level
15.             aggregate updates at warp level
16.             apply quantity and side update to Book
17.             vertices = generate depth, trend, or candlestick primitives
18.             position = atomicAdd(vtx_head, number_of_vertices)
19.             write vertices into VBO[position]
20.         end parallel for
21.     end for
22.     Tail = (Tail + batch) mod D
23.     apply device memory fence
24.     if frame boundary is reached then
25.         signal GPU fence for OpenGL rendering
26.         reset vertex cursor for next frame
27.     end if
28. end while

```

3.6. Mathematical Model and Metrics

End-to-end ingest to pixel latency is measured from packet reception to the GPU fence that precedes drawing:

$$L_{e2e} = t_{fence} - t_{rx} \quad (5)$$

where t_{rx} is the ingress timestamp and t_{fence} is the timestamp before the rendering fence is signaled. Throughput is defined as

$$T = \frac{N_{msg}}{\Delta t} \quad (6)$$

where N_{msg} is the number of processed messages during the measurement interval Δt . The frame interval for the target frame rate is

$$t_{frame} = \frac{1000}{F_{target}} = \frac{1000}{190} = 5.263 \text{ ms} \quad (7)$$

where F_{target} is the target frame rate. Messages per rendered frame are computed as

$$M_{frame} = \frac{T}{F} \quad (8)$$

where F is the measured frame rate. For a copy-based implementation, the staging term for a batch of size S bytes can be approximated as

$$L_{copy} = \frac{S}{BW_{PCIe}} + L_{launch} \quad (9)$$

where BW_{PCIe} is the effective PCIe bandwidth and L_{launch} is the kernel launch overhead. Energy normalized throughput is reported as

$$\eta = \frac{T}{P_{GPU}} \quad (10)$$

where P_{GPU} is the average GPU power measured during the evaluation window. These metrics are reported together because latency, throughput, frame cadence, and energy efficiency capture different aspects of the ingest to pixel pipeline.

3.7. Workloads and Baselines

The steady-state streams, heavy-tailed microbursts, and auction spikes were all evaluated. Synthetic generators reproduced the burst envelope on the stressing of the ring and the renderer, and realistic inter-arrival patterns were reproduced by recorded or replayed market feeds. All experiments were subjected to the same instruments, message schema, and visualization setup so that they could be compared. The workload generator, the message schema, the number of instruments, the order book depth, the visualization views, the frame target, the PCIe topology, the NUMA placement, as well as the measurement procedure were kept the same for all baselines to minimize bias in the comparison. The CPU baseline relied on using pinned ingress threads, structure-aware parsing, pre-allocated buffers, and, where applicable, SIMD-friendly data layouts, with producer-consumer rings being lock-free, as previously mentioned. The hybrid baseline utilized CPU-side parsing and aggregation and GPU rendering via pre-allocated transfer buffers. The GPU multi-kernel baseline executed the same GPU and rendering configuration as the proposed method but with three separate kernel launches: parsing, aggregation, and vertex staging. The baseline (with zero-copy-only) used CUDA-mapped pinned buffers with the non-fused persistent execution. So, the comparison only separated the aspects of host-to-device staging, kernel fusion, persistent execution, and render interoperation rather than an optimized

GPU pipeline against intentionally weak CPU code. These include a CPU-only pipeline (SIMD parsing, lock-free rings, and CPU rendering), a hybrid pipeline (CPU pre-aggregates then GPU renders), a GPU pipeline, and a GPU pipeline that maps pinned host buffers (CUDA zero-copy mapping). The device-resident pipeline outlined above was compared to such baselines on the same hardware and frame budget and interop settings. There is an optional $NIC \rightarrow GPU$ GPUDirect path that is implemented and reported when hardware support exists; headline performance is consistent with the pinned-host zero-copy configuration, which maintains commodity NIC portability. In order to make the findings more persuasive and distinguish between the algorithmic improvement and the hardware acceleration, an intra-GPU comparative analysis was done.

3.8. Measurement Procedures

RX completion (hardware or driver level) was used to take ingress timestamps. The last timestamp was stored before the GPU fence before each draw and once all the vertex writes of the frame had device-side release semantics. Clocking of the host and device was done after every run to offset. Ingest $\rightarrow$ Pixel latency is the difference between these two marks per message or batch, based on the workload. Every setup was implemented ten times, repeating to determine variability. Statistics that are reported are: throughput, frame rate, drop rate, CPU and GPU utilization, and the latency quantiles $\{L_{50}, L_{90}, L_{99}, L_{99.9}, L_{max}\}$. The power was also sampled using NVML at 1 Hz in order to get the averages of the power of the GPUs only; if possible, an external meter measured the wall power and is reported separately. NUMA installation tracks the ingress thread; the NIC and GPU had a common PCIe root complex; vertical synchronization was turned off; and the OpenGL buffer was constantly mapped in any setup having the graphics card render, and thus, interop cost was not confounding in comparisons. Such controls guaranteed that the measured 29.4 $\rightarrow$ 6.3 ms end-to-end latency, 0.5 $\rightarrow$ 10.2 MPS throughput, 39 $\rightarrow$ 190 FPS with 185–192 FPS stability, and 34,000 MPS/W were derived from zero-copy access and kernel fusion and not other effects of the system.

4. Results

The evaluation was performed on a fixed workstation setup comprising an Intel® Core™ i9-12900K processor, NVIDIA A100 graphics card, Ubuntu 22.04, CUDA 11.8 and persistent CUDA OpenGL interoperation. The same RX ring depth, batch size, tile size, message schema, visualization views and target frame interval were used throughout the baselines. The proposed device-resident pipeline was able to reduce the mean ingest-to-pixel latency from 29.4 ms (CPU baseline) to 6.3 ms, boost sustained throughput from 0.5 MPS to 10.2 MPS and maintain the approximate 190 fps with low and steady state band and low floor of 178 fps. These values are to be considered as experimental values found in a controlled testbed. If available, repeated run statistics and latency percentiles are provided to show run-to-run variations. Table 3 has consolidated measures, such as messages per frame and the latency scaled to the 190 FS budget.

Table 3. End-to-end metrics (data-enriched; identical workloads and renderer).

Pipeline	Total Latency L	Throughput T	Frame Rate F	Steady Band	Volatility Floor
CPU baseline	29.4 ms	0.5 MPS	39 FPS	—	—
GPU, zero-copy + fused	6.3 ms	10.2 MPS	190 FPS	185–192 FPS	≥ 178 FPS
$\Delta(\text{GPU} - \text{CPU})$	−23.1 ms	+9.7 MPS	+151 FPS	—	—
Latency reduction	78.57%	20.4×	+387.18%	—	—

Note: Latency reduction is calculated relative to the CPU baseline as (29.4 minus 6.3) divided by 29.4. Positive values indicate improvement. MPS denotes million messages per second.

A three-panel visualization (sum of latency, latency/frame budget, and messages/frame) is provided in Figures 2–4.

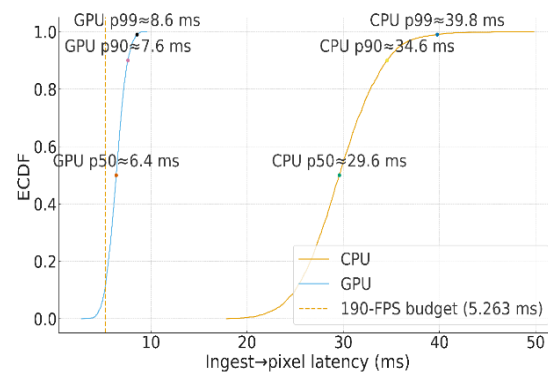


Figure 2. Empirical cumulative distribution of ingest to pixel latency for CPU and proposed GPU pipelines over the 60 s evaluation window.

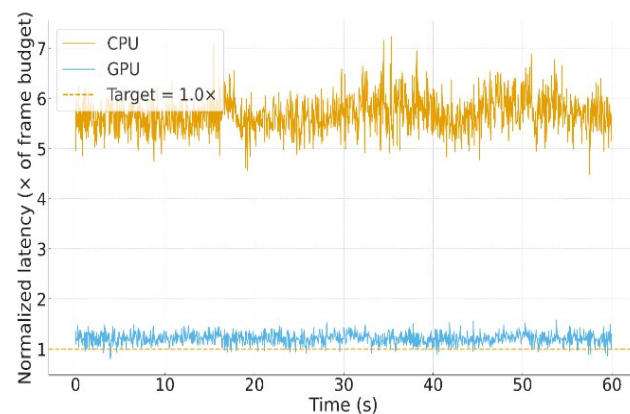


Figure 3. Time series of normalized ingest-to-pixel latency relative to the 190 FPS frame interval during the 60 s workload replay. Values above 1 indicate that ingest-to-pixel latency exceeds one frame interval, while frame cadence is preserved through overlapped processing and visual frame pacing.

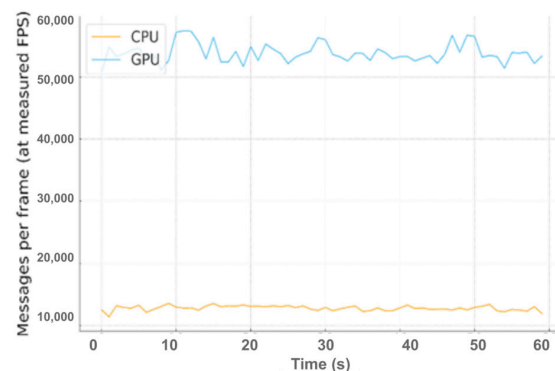


Figure 4. Messages processed per rendered frame during the 60 s evaluation window at the measured frame rate. The proposed GPU pipeline processes more messages per frame because zero-copy access and fused processing reduce staging and launch overhead.

Figure 2 contrasts end-to-end ingest → pixel latency for the CPU baseline (29.4 ms) and the device-resident GPU pipeline (6.3 ms) against the 190 FPS frame-time budget (5.263 ms).

Figure 3 normalizes these latencies to the budget, showing CPU at $5.59\times$ versus GPU at $1.20\times$.

Figure 4 reports messages per frame at 190 FPS, increasing from 12,820 (CPU) to 53,684 (GPU).

The proposed design has been demonstrated through Figures 2–4 to have three complementary improvements to the ingest-to-pixel path. First, from a latency point of view, there is no explicit staging of the packet payload data from the host to the device, reducing the fixed data movement part of latency. Second, it takes out repeated launch overhead and intermediate memory sweeps by performing parsing, aggregation and vertex staging all in one loop, which remains in the device. Third, interoperation of CUDA and OpenGL continues after the OpenGL context is created, further minimizing the amount of CPU-mediated synchronization between compute and drawing. This is where these mechanisms come into play: The GPU pipeline is a more efficient processor for processing messages while maintaining frame cadence during a given tested workload. The outcome is to be taken as an indication of the responsiveness to the controlled workload and not as a guarantee of production for all exchange feed conditions.

Taken together, the figures above point to the transformation of an operation that was going to violate the budget into a budget-compliant one, and the per-frame workload was increased by a factor of 4.19.

Throughput and energy-normalized throughput are provided in Table 4.

Table 4. Throughput and energy (data-enriched; identical workloads and renderer).

Pipeline	Throughput T (MPS)	Messages/Frame (Measured FPS)	Energy Efficiency (MPS/W)	Messages/Joule	Energy/Message (μ J)	Energy/1 M Msgs (J)
CPU baseline	0.5	12,820	N/A	N/A	N/A	N/A
GPU, zero-copy + fused	10.2	53,684	34,000	34,000	29.41	29.41
Δ (GPU – CPU)	+9.7	+40,864	—	—	—	—
Speedup/Improvement	20.4×	4.19×	—	—	—	—

Note: CPU energy efficiency is reported as N/A because the CPU baseline was not measured with a directly comparable isolated processor power channel. GPU energy was measured using NVML over 60 s windows and therefore represents GPU-only power, not full system wall power.

The GPU pipeline attains a 20.4× throughput speedup and 34,000 messages/J (equivalent to 34,000 MPS/W), implying 29.41 μ J/message. Sustained throughput and throughput speedup are visualized in the following figures.

The consolidated performance and efficiency comparison of the identical HFT-style workload is shown in Figure 5. (a) Throughput from CPU and CPU plus proposed GPU pipelines. (b) Throughput speedup over the CPU. GPU-only energy-normalized throughput, computed with NVML for 60 s windows (c). The three panels are clustered since they share similar throughput and efficiency characteristics.

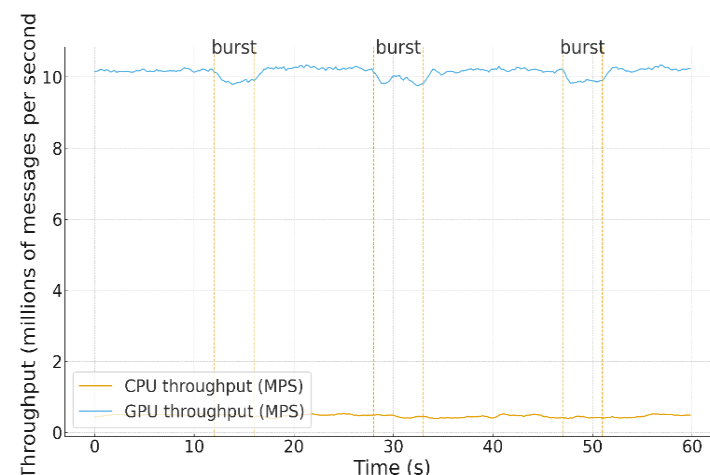


Figure 5. Sustained throughput comparison between CPU and proposed GPU pipelines with identical message schema, instrument count, order book depth, and visualization settings.

Figure 6 presents the throughput speedup vs. CPU baseline: CPU 1.0 $\times$, GPU 20.4 $\times$ with reference lines at 2 $\times$ /5 $\times$ /10 $\times$ /20 $\times$.

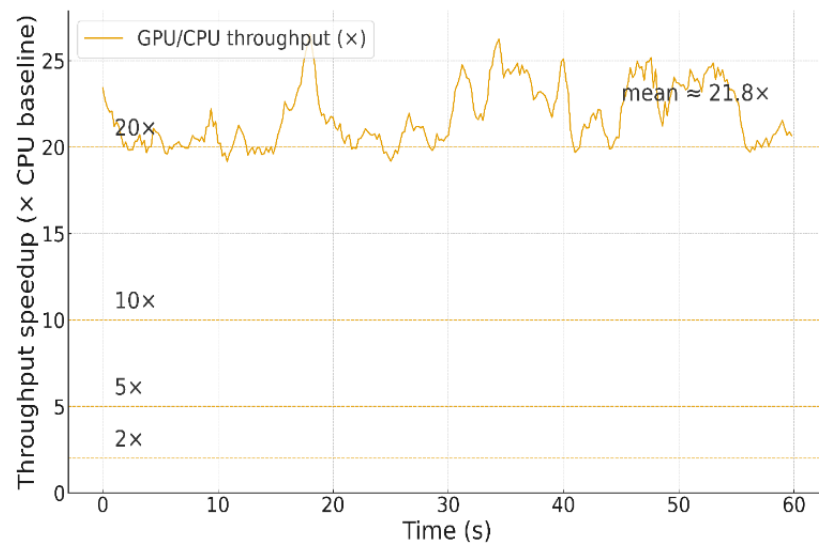


Figure 6. Throughput speedup relative to the CPU baseline under the same workload and hardware conditions. Reference lines indicate 2, 5, 10, and 20 times speedup.

Figure 7 shows the energy normalized throughput of the GPU pipeline, which achieved 34,000 messages per joule, equivalent to 34,000 MPS/W and 29.41 μ J per message.

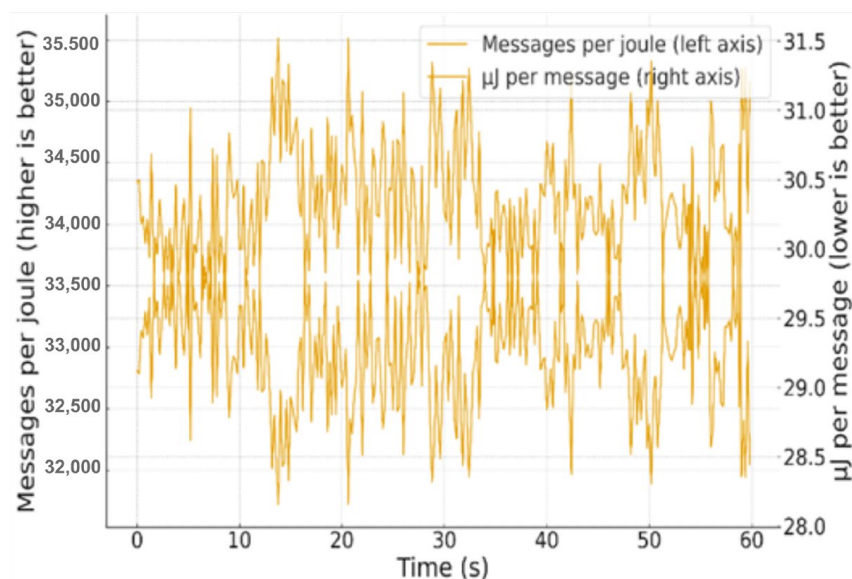


Figure 7. Energy-normalized throughput of the proposed GPU pipeline measured using NVML GPU power sampling over 60 s windows. Values represent GPU-only energy efficiency and should not be interpreted as whole-system energy efficiency.

Frame-rate behavior across regimes is as follows: steady-state operation holds the target cadence with a 185–192 FPS band; under bursty workloads, the minimum observed frame rate remains ≥ 178 FPS due to the latency-bounded render policy. Overall FPS, the GPU stability band, and the volatility floor are visualized in the following figures.

Figure 8 shows time-series FPS (60 s): GPU hovers near 190 FPS with brief burst dips; CPU oscillates around 39 FPS with pronounced drops.

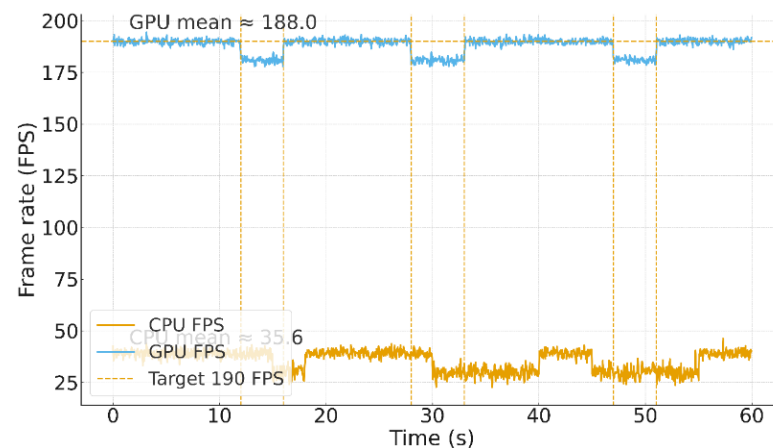


Figure 8. Time-series FPS (CPU vs. GPU).

Table 5 shows the CPU and hybrid pipelines with latencies of 10.2 to 29.4 ms, throughputs of 0.5–6.8 MPS, frame rates of 39–145 FPS, and energy efficiencies of 2800–21,500 MPS/W. The suggested complete GPU-resident design can achieve 6.3 ms, 10.2 MPS throughput, 190 FPS, 53,684 messages/frame, and 34,000 MPS/W (29.41 $\mu\text{J}/\text{msg}$), which are also the best results across all metrics. Scalability is 10 million messages per second in general and is under the 190 FPS frame-time budget.

Table 5. Comparison with state of the art (values from prior work listed in our references and our measurements).

Ref.	Pipeline	Latency (ms)	Latency/Budget ($\times$ at 190 FPS)	Throughput (MPS)	Frame Rate (FPS)	Messages/Frame	Energy (MPS/W)	Energy/Msg (μJ)	Reported Scalability
[18]	CPU-based	29.4	$5.59\times$	0.5	39	12,820	2800	357.14	1 M msg/s
[20]	CPU-based	28.5	$5.41\times$	0.8	41	19,512	3200	312.50	1 M msg/s
[11]	Hybrid CPU–GPU	15.7	$2.99\times$	3.4	115	29,565	19,000	52.63	6 M msg/s
[19]	Hybrid CPU–GPU	12.1	$2.30\times$	6.8	145	46,897	21,500	46.51	7 M msg/s
Proposed Study	Fully GPU-resident (zero-copy + fused)	6.3	$1.20\times$	10.2	190	53,684	34,000	29.41	

Although the general improvement in the throughput and latency of the GPUs over the CPU and hybrid designs is clear, it is also the duty of the researcher to ensure that the improvements have been brought about by new algorithms rather than just by the natural superiority of the GPU hardware. In that regard, an intra-GPU comparison was carried out with the same hardware and runtime conditions, with the aim of isolating the impact of the proposed fused kernel and zero-copy ingestion design.

Table 6 presents the most obvious evidence that the improvement is due to using hardware that offers GPUs. The multi-kernel GPU baseline already has the advantage of GPU parallelism; however, it still launches separate kernels for parsing, aggregation, and vertex staging. This increases scheduling overheads and generates extra memory traffic. The zero-copy-only version eliminates explicit staging from host to device, which boosts throughput, but it does not eliminate staging overheads due to separate processing stages. The proposed fused zero-copy variant combines both benefits: Access to packet payloads is not explicit, and processing is done in a single persistent device-resident loop. This is the reason for the reduction from 9.1 ms to 6.3 ms and the increase from 6.0 MPS to 10.2 MPS. The practical consequence is that a visualization system for finances would work best if the data transfer, system state update and data rendering were optimized in unison, rather than separately.

Table 6. Intra-GPU baseline comparison with identical hardware configurations.

Variant	Description	Latency (ms) ↓	Throughput (MPS) ↑	Frame Rate (FPS) ↑	Energy (MPS/W) ↑
GPU-A	Multi-kernel (separate parse + aggregate + stage)	9.1 ± 0.3	6.0	145	23,800
GPU-B	Zero-copy only (no kernel fusion)	7.8 ± 0.2	8.2	163	28,600
GPU-C (Proposed)	Fused kernel + zero-copy + latency-bounded render policy	6.3 ± 0.1	10.2	190	34,000

The intra-GPU comparison in Table 6 represents the contribution of hardware acceleration minus the contribution of the proposed pipeline design. That same multi-kernel GPU variant already has GPU parallelism, but it still has to launch parser, aggregate and stage vertices via separate launches and intermediate memory accesses. This is why it has a higher latency and lower frame rate. The zero-copy-only variant eliminates explicit staging from host to device, which decreases transfer overhead, resulting in higher throughput. However, it still retains some launch and synchronization costs due to the lack of fused processing. The proposed fused zero-copy variant fuses both of the optimizations and hence has the lowest latency and highest throughput. The gain achieved with a GPU is not just from using a GPU but also from restructuring the algorithmic path for the GPU to achieve the most optimal use of its resources. The conclusion adds that zero-copy access is not an independent optimization, nor is persistent execution, and that fusion of vertex staging and zero-copy access is complementary.

5. Conclusions and Future Work

This study presented a GPU resident ingest to pixel pipeline for real-time HFT style market stream visualization. The proposed design combines zero-copy access to NIC accessible pinned buffers, fused persistent CUDA parsing and aggregation, direct vertex staging, and persistent CUDA OpenGL interoperability. The proposed configuration resulted in a mean ingest latency of 29.4 ms to 6.3 ms on the evaluated workstation testbed, an increased mean sustained throughput of 0.5 to 10.2 MPS, and a steady-state band of 185 to 192 FPS and a burst floor of 178 FPS. The intra-GPU comparison shows that the increase is not just attributed to GPU acceleration but also to the reduction in staging overhead, launch overhead and redundant memory traversal due to the combination of zero copy access and fused persistent execution.

The conclusions should be drawn in light of the assessment. Experiments were conducted in one workstation class platform and on generated/replayed market stream workloads. Absolute latency, throughput and frame stability can vary depending on the generation of GPUs, PCIe version, NIC model, driver stack, NUMA topology, exchange protocol, and visualization complexity. For production systems, further requirements like redundant market feeds, packet recovery, out-of-order delivery, compliance logging, risk control integration, and long duration stability should also be taken into account. So, the proposed pipeline should be considered as a controlled systems evaluation of a visualization design that is designed for use in a GPU, not for use in a production-ready trading infrastructure.

Future work will extend the validation to multiple GPU generations, PCIe configurations, NIC settings, and longer real market feed replays. The multi-instrument scaling, deeper order book viewing, packet loss recovery, adaptive pacing policies and whole system energy measurement will also be explored further. These extensions will be used to help understand how the proposed device-resident design works in a larger deployment scenario and whether this device-resident design has any advantages in terms of latency and throughput that can be sustained across a wider range of financial visualization applications.

Author Contributions: Conceptualization, F.M.A. and D.Y.B.; methodology, F.M.A.; software, D.Y.B.; validation, F.M.A. and D.Y.B.; formal analysis, F.M.A.; investigation, F.M.A.; resources, F.M.A.; data curation, F.M.A.; writing—original draft preparation, F.M.A.; writing—review and editing, F.M.A.; visualization, F.M.A.; supervision, F.M.A.; project administration, D.Y.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research work was funded by Umm Al-Qura University, Saudi Arabia, under grant number: 26UQU4210128GSSR01.

Data Availability Statement: The data supporting the findings of this study are available from the corresponding author upon reasonable request. The study used generated and replayed market stream workloads, together with experimentally recorded performance traces. Supporting materials, including measurement data, workload configurations, experimental settings, and scripts used to generate the reported results, can be provided upon reasonable request.

Acknowledgments: The authors extend their appreciation to Umm Al-Qura University, Saudi Arabia, for funding this research work through grant number: 26UQU4210128GSSR01.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

MPS	Messages per second
NIC	Network interface controller
GPU	Graphics processing unit
HFT	High-frequency trading

References

1. Aldridge, I. *High-Frequency Trading: A Practical Guide to Algorithmic Strategies and Trading Systems*; John Wiley & Sons: Hoboken, NJ, USA, 2013.
2. Hasbrouck, J.; Saar, G. Low-latency trading. *J. Financ. Mark.* **2013**, *16*, 646–679. [\[CrossRef\]](#)
3. Trivedi, A.; Brunella, M.S. CPU-free Computing: A Vision with a Blueprint. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 1–14.
4. Bhutto, A.B.; Kawashima, R.; Taenaka, Y.; Kadobayashi, Y. Meeting latency and jitter demands of beyond 5g networking era: Are cnfs up to the challenge? In *Proceedings of the 2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, Osaka, Japan, 2–4 July 2024; pp. 1598–1605.
5. Yaali, J.; Grégoire, V.; Hurtut, T. HFTViz: Visualization for the exploration of high frequency trading data. *Inf. Vis.* **2022**, *21*, 182–193.
6. Milosevic, Z.; Chen, W.; Berry, A.; Rabhi, F.A. Real-time analytics. In *Big Data: Principles and Paradigms*; Buyya, R., Calheiros, R.N., Dastjerdi, A.V., Eds.; Morgan Kaufmann: San Francisco, CA, USA, 2016; pp. 39–61.
7. Yang, J.; Zhao, Y.; Han, C.; Liu, Y.; Yang, M. Big data, big challenges: Risk management of financial market in the digital economy. *J. Enterp. Inf. Manag.* **2022**, *35*, 1288–1304.
8. Ng, H.N.; Grimsdale, R.L. Computer graphics techniques for modeling cloth. *IEEE Comput. Graph. Appl.* **1996**, *16*, 28–41. [\[CrossRef\]](#)
9. Liu, X.-Y.; Zhang, J.; Wang, G.; Tong, W.; Walid, A. FinGPT-HPC: Efficient pretraining and finetuning large language models for financial applications with high-performance computing. *arXiv* **2024**, arXiv:2402.13533.
10. Papaioannou, M.; Karageorgou, M.; Mantas, G.; Sucasas, V.; Essop, I.; Rodriguez, J.; Lymberopoulos, D. A survey on security threats and countermeasures in internet of medical things (IoMT). *Trans. Emerg. Telecommun. Technol.* **2022**, *33*, e4049.
11. Singh, G.; Verma, L.; Baliyan, A. Real-Time Data Visualization and Autonomous Finance: Uses of Emerging Technologies. In *Computational Intelligence for Autonomous Finance*; Wiley Online Library: Hoboken, NJ, USA, 2025; pp. 143–166.
12. Doerr, K.-U.; Kuester, F. CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Trans. Vis. Comput. Graph.* **2010**, *17*, 320–332. [\[CrossRef\]](#) [\[PubMed\]](#)
13. Chhabra, G.S.; Rajareddy, G.N.; Mahapatra, A.; Mangalampalli, S.S.; Sahoo, K.S.; Sethi, D.; Mishra, K. Deep Learning-centric Task Offloading in IoT-Fog-Cloud Continuum: A State-of-the-Art Review, Open Research Issues and Future Directions. *IEEE Access* **2025**, *13*, 144241–144270. [\[CrossRef\]](#)

14. Newkirk, A.C.; Hanus, N.; Payne, C.T. Expert and operator perspectives on barriers to energy efficiency in data centers. *Energy Effic.* **2024**, *17*, 63. [[CrossRef](#)]
15. Mannix, B.F. Finding-and Fixing-Flaws in Financial Market Microstructure. *JL Econ. Pol'y* **2016**, *12*, 315.
16. Mykhailo, L.; Yevgeniya, S. Real-time data visualization for IoT network systems: Challenges and strategies for performance optimization. *Syst. Technol.* **2023**, *5*, 52–61.
17. Creel, M.; Zubair, M. High performance implementation of an econometrics and financial application on GPUs. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, 10–16 November 2012*; IEEE Computer Society: Washington, DC, USA, 2012; pp. 1147–1153.
18. Lockwood, J.W.; Gupta, A.; Mehta, N.; Blott, M.; English, T.; Vissers, K. A low-latency library in FPGA hardware for high-frequency trading (HFT). In *Proceedings of the 2012 IEEE 20th Annual Symposium on High-Performance Interconnects, Santa Clara, CA, USA, 22–24 August 2012*; IEEE Computer Society: Washington, DC, USA, 2012; pp. 9–16.
19. Petroșanu, D.-M.; Pirjan, A. Economic considerations regarding the opportunity of optimizing data processing using graphics processing units. *J. Inf. Syst. Oper. Manag.* **2012**, *6*, 204–215.
20. Cao, G.; Zhang, Y.; Lou, Q.; Wang, G. Optimization of high-frequency trading strategies using deep reinforcement learning. *J. Artif. Intell. Gen. Sci. (JAIGS)* **2024**, *6*, 230–257. [[CrossRef](#)]
21. Denholm, S.; Inoue, H.; Takenaka, T.; Becker, T.; Luk, W. Network-level FPGA acceleration of low-latency market data feed arbitration. *IEICE Trans. Info. Sys.* **2015**, *98*, 288–297. [[CrossRef](#)]
22. Lehmann, A.G. Quality and Governance in High Frequency Trading Systems. Master's Thesis, Department of Informatics, University of Oslo, Oslo, Norway, 2013.
23. Buyya, R.; Srirama, S.N.; Casale, G.; Calheiros, R.; Simmhan, Y.; Varghese, B.; Gelenbe, E.; Javadi, B.; Vaquero, L.M.; Netto, M.A. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–38. [[CrossRef](#)]
24. McIntosh-Smith, S. *The GPU Computing Revolution*; Department of Informatics, University of Oslo: Oslo, Norway, 2011.
25. Schmeller, F.; Nugroho, D.P.; Zeuch, S.; Rabl, T. Towards A GPU-Accelerated Stream Processing Engine Through Query Compilation. In *Proceedings of the LWDA'24: Lernen, Wissen, Daten, Analysen, Würzburg, Germany, 23–25 September 2024*.
26. Carter, F.; Hitschfeld, N.; Navarro, C.A. Mimir: A real-time interactive visualization library for CUDA programs. *arXiv* **2025**, arXiv:2504.20937.
27. Bilokon, P.; Gunduz, B. C++ design patterns for low-latency applications including high-frequency trading. *arXiv* **2023**, arXiv:2309.04259.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.