High Frequency Trading System

Vrushang Prakashkumar Anand
Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, United States
vpanand@uci.edu

Jonathan Yannick Emanuel Ajavon
Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, United States
jajavon@uci.edu

Hankang Li
Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, United States
hankanl@uci.edu

Ali Sao
Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, United States

Ruiqi Peng
Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, United States

***Abstract*—This document presents the development of a high-frequency trading (HFT) system that incorporates FPGA-based hardware acceleration, reinforcement learning, and red-black trees for fast order processing and decision-making. The system features a Tensor Processing Unit (TPU) implemented directly on an FPGA that runs a lightweight neural network for intelligent order placement, a high-throughput MultiQueue order matching engine capable of processing 75 million orders per second, and a software backend using lock-free Red-Black Trees for large-scale order management on the host computer. This document outlines our system architecture, objectives, tools used, and experimental results.**

***Index Terms*—high-frequency trading, reinforcement learning, FPGA, neural networks, data structures, system design, hardware acceleration, systolic array, MultiQueue**

I. Introduction

High-frequency trading (HFT) requires systems that can process massive amounts of market data with extremely low latency. Standard CPU-based trading engines struggle with performance due to limitations in sequential execution and memory access. To overcome this, our Senior Design Project incorporates FPGA acceleration for parallel order matching and pipelined execution with on-chip memory for massive performance boosts. This method allows our system to achieve maximum performance and low-latency order matching.

Furthermore, a Tensor Processing Unit (TPU) is implemented directly on the FPGA to run a lightweight neural network that determines which orders are most profitable and should be cached in Block RAM (BRAM). Instead of sending order book data to the CPU for inference, the entire decision pipeline is executed in hardware, eliminating software scheduling overhead and reducing latency. The BRAM acts as a cache for the "hottest" orders, enabling fast lookup and execution of trades. With the combination of FPGA acceleration, efficient software infrastructure, and adaptive machine learning algorithms, our system provides an innovative solution in maintaining speed, reliability, and scalability in modern financial trading systems.

II. Project Overview and Timeline

Our High-Frequency Trading system was developed over two academic quarters (Fall 2024 and Winter 2025) as a Senior Design Project. The overarching goal was to build a complete, end-to-end HFT system combining FPGA hardware acceleration, machine learning-driven order prioritization, and a real-time web-based user terminal. Table I summarizes the key milestones and accomplishments across both quarters.

TABLE I
Project Timeline and Accomplishments

Quarter	Milestone	Outcome
Fall 2024	Thread-safe Red-Black Tree	Completed; $O(\log n)$ verified
	FPGA BRAM data structures	Initial MultiQueue prototype built
	Neural network on FPGA	Initialized; single forward pass in 50 ns
Winter 2025	MultiQueue finalized	75M orders/sec; 2-cycle deterministic match
	CAM controller	Dynamic symbol indexing implemented
	TPU integration	20 ns inference at 200 MHz; 63% accuracy
	WebSocket UI	Real-time order book and trade tracking live
	System integration	SPI link between host, TPU FPGA, and Order Matching FPGA

III. Objectives

A. Overall Objectives

- Design and build a complete high-frequency trading system capable of executing order matching in real time.
- Deploy a reinforcement learning model on an FPGA acting as a TPU that dynamically calculates which orders should be cached in FPGA BRAM for ultra-low-latency access.
- Develop a scalable software backend using lock-free Red-Black Trees for large-scale order storage on the host computer.
- Achieve a trade matching throughput of 75 million orders per second using a novel MultiQueue FPGA architecture.

B. Objectives for Fall Quarter

The following objectives were set for Fall 2024. All three were successfully completed:

- ✓ **[Completed]** Implement thread-safe, lock-free data structures (Red-Black Trees) for order storage on the CPU. Unit tests confirmed $O(\log n)$ search, insertion, and deletion with correct balancing.
- ✓ **[Completed]** Initialize the neural network inference pipeline on the FPGA. A single forward pass was achieved in 50 ns.
- ✓ **[Completed]** Create efficient BRAM data structures for order storage on the FPGA. An initial MultiQueue prototype was built and validated.

C. Objectives for Winter Quarter

The following objectives were set for Winter 2025. All major objectives were completed; backtesting optimization remains ongoing:

- ✓ **[Completed]** Finalize and test the MultiQueue BRAM structure — achieves 75 million orders/second with deterministic 2-cycle matching.
- ✓ **[Completed]** Set up a WebSocket connection for real-time market data communication between the React UI and FPGA backend.
- ✓ **[Completed]** Integrate the TPU FPGA into the full system pipeline via SPI; inference runs at 200 MHz with a 20 ns decision latency.
- ✓ **[Completed]** Implement Content-Addressable Memory (CAM) controller for dynamic symbol indexing within the MultiQueue.
- ○ **[Ongoing]** Full backtesting on historical data and further optimization of the overall decision-making pipeline. Preliminary results show 63% order capture accuracy; continued tuning of the RL agent is planned.

IV. System Architecture

The system consists of three primary components: the host computer, the TPU FPGA, and the Order Matching FPGA. The host computer manages over 100,000 orders using Red-Black Trees and connects to the FPGAs via parallel SPI, enabling low-latency and high-throughput data transfer between the CPU software and the specialized hardware. The TPU FPGA runs the neural network to determine order priority, and the Order Matching FPGA executes high-speed trade matching entirely in on-chip BRAM.

V. Software Infrastructure

For order storage on the host system, we use Red-Black Trees to store and manage all orders. Since the on-chip memory available on the FPGA is limited (approximately 32 KB), the majority of orders reside in the software infrastructure on the host computer, while only the "hottest" trades are cached on the FPGA for ultra-low-latency access. To handle high data throughput, we implemented Red-Black Trees and Radix Tries, which are highly optimized for cache efficiency. These data structures use a lock-free implementation to manage concurrent access from different strategy threads, allowing us to scale calculations across multiple CPU threads.

We chose Red-Black Trees specifically because of the balanced structure they provide, guaranteeing $O(\log n)$ time complexity for all operations. The orders are stored in the Red-Black Trees based on a rank determined by factors such as how frequently the corresponding stock is traded or referenced. This rank determines which orders are promoted to FPGA memory. The rank depends on two main factors: the frequency of the stock and the nice value given by the RL agent. The formula is as follows:

$$\text{Rank} = \frac{\text{Frequency} \times C}{\text{Nice Value Weight}}$$

where C is a constant scaling factor.

VI. Trade Storage and Matching

To efficiently store and match incoming orders, we use a novel MultiQueue architecture implemented on the FPGA [1]. The architecture supports four operations: INSERT, REMOVE, INSREM, and NOP. It builds upon the Systolic Array architecture: when inserting an item into our priority queue, all items with lower priority are pushed back using Sorting Cells. Rather than storing data in shift registers, we utilize BRAM blocks, trading the ability to read from multiple queues simultaneously for significantly reduced usage of Configurable Logic Blocks, since those are only used for Sorting Cells.

The key components of the Order Matching Engine are:

- **Memory**: The MultiQueue architecture uses Block RAM to store 512 symbols, each with a 24-order deep Min-Queue, providing deterministic 2-cycle read and write performance.
- **Dynamic Indexing**: A Content-Addressable Memory (CAM) controller acts as a lookup table for symbols stored in the MultiQueue. It automatically clears the oldest symbol when full, enabling dynamic symbol management without software intervention.
- **Pipelining**: With extensive pipelining, CAM lookup and order matching logic execute in parallel with MultiQueue read/write. Incoming orders are deterministically matched

within 2 clock cycles, leveraging BRAM for predictable latency.

For our FPGA, the Artix-7 XC7A35T supports up to 100 blocks of 18 Kb BRAM, theoretically allowing up to 1,024 priority queues each of depth 100, storing 18 bits of data per item, as shown in the following diagram.

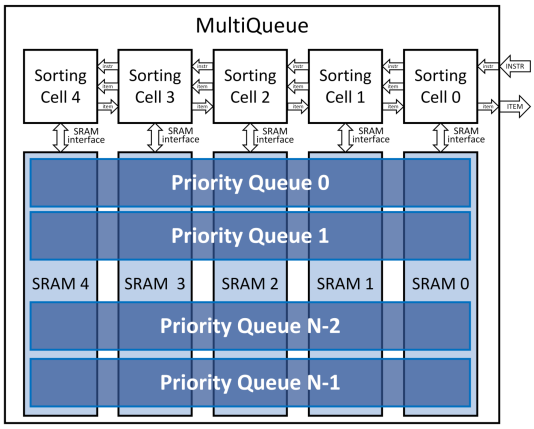

Fig. 1. **MultiQueue architecture.**

VII. Tensor Processing Unit (TPU)

The TPU accelerates trading decisions by running a lightweight neural network directly on the FPGA. Instead of sending order book data to the CPU, inference is performed entirely in hardware, reducing latency and eliminating software scheduling overhead.

- **Precision**: Uses 32-bit fixed-point (Q2.29) arithmetic for highly efficient integer-based computation while maintaining accuracy [2].
- **Logic**: The decision engine implements the pipeline MatMul $\rightarrow$ Tanh $\rightarrow$ MatMul $\rightarrow$ Softmax, acting as an assembly line that processes orders layer by layer.
- **Systolic Array**: A hardware grid processes entire chunks of the neural network at once, enabling high-throughput parallel matrix operations.
- **Optimizations**: Complex math operations (Tanh and Softmax) are approximated using on-chip Look-Up Tables (LUT) to reduce computational latency and eliminate floating-point overhead.
- **Performance**: With math and execution pipelining, the system produces a 20 ns decision latency at a clock frequency of 200 MHz (5 ns per stage).

VIII. Reinforcement Learning

We use a Reinforcement Learning (RL) model to determine the nice value for each trade. Specifically, the Proximal Policy Optimization (PPO) algorithm is used to train our RL agent [3]. The policy is a neural network that takes multiple aspects of the order into account:

- **Frequency**: How often the stock is traded or referenced, gauging popularity.
- **Volume**: The number of shares involved in trades, with larger share volumes offering greater profit potential.
- **Buy/Sell Ratio**: Measures market liquidity by comparing the number of buy and sell orders. A more balanced ratio indicates easier tradability.

The output of this policy is a list of probabilities for the nice values in the range $[-10, 10]$, each carrying a specific weight to determine the rank of the trade. Based on the output and trade performance, a reward is assigned depending on whether the order was in FPGA memory when it was matched and what the rank of the trade was:

$$\pi_\theta(a \mid s) = \Pr\left(\text{Nice Value} = a \;\middle|\; s\right)$$

$$R(s,a) = \begin{cases} \alpha \cdot \text{Rank}(s) & \text{if the order was in FPGA memory,} \\ \beta \cdot \text{Rank}(s) & \text{otherwise,} \end{cases}$$

where s is the state vector representing features of the order, $a \in [-10, 10]$ is the nice value, and $\alpha > \beta$ are constants that weight the reward higher when the trade was matched from FPGA memory.

IX. User Interface (UI)

To interact with the system, we created a modern web-based user interface (User Terminal) for the HFT system. The UI was first prototyped in Figma to refine the design, then implemented in React for real-time functionality. Our design emphasizes responsiveness, clarity, and usability to ensure traders can make fast, informed decisions.

The main features include a real-time table of stock prices, interactive graphs showing price movements, a visual order book to monitor market depth, and buy/sell buttons with live profit tracking.

Real-time updates are pushed through WebSockets, allowing constant two-way communication between the React front end and the FPGA back end. When a trade execution occurs, the UI reflects the updated position and adjusts profit tracking without delay.

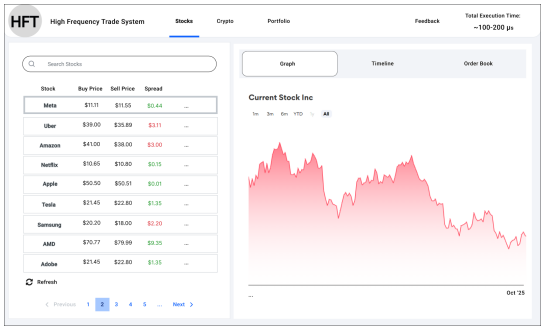

Fig. 2. Web User Interface (User Terminal)

Setup Details of Project

Software

- C Programming Environment: GCC 13.2.0
- Python Environment: Python 3.13
  - Stable-Baselines3
  - Gymnasium
  - NumPy 2.3
  - PyTorch 2.2.0
- Verilog: Xilinx Vivado 2023.1
- Docker Engine 29.1.2
- Figma: 2025 web release
- Node.js 20.9.0

Hardware

- Cmod A7-35T: Breadboardable Artix-7 FPGA Module (×2: one for TPU, one for Order Matching)

X. Security Considerations

Because high-frequency trading systems interact with financial markets, security is critical.

A. Network Security

All communication between the UI and backend uses TCP sockets. Future versions should implement TLS encryption and authentication to prevent spoofed trading commands or packet injection.

B. FPGA Firmware Integrity

Unauthorized modification of FPGA bitstreams could allow malicious trading strategies. Bitstream encryption and secure boot mechanisms should be used in production deployments.

C. Market Manipulation Prevention

Automated trading systems may unintentionally enable strategies that resemble market manipulation. Risk control logic and order rate limits should be implemented.

D. Infrastructure Security

If deployed in production, the system should run within a secure trading infrastructure with restricted access and logging of all trading actions.

XI. Standards Used

- **SPI (Serial Peripheral Interface)**: Used for communication between the host computer and the FPGAs, providing low-latency and high-throughput parallel data transfer.
- **TCP**: While a low-latency UDP interface was initially planned to reduce protocol overhead between the CPU and user terminal, integration constraints required falling back to TCP to ensure stable and reliable data transmission during testing.
- **PPO**: Reinforcement learning algorithm chosen for its stability and low variance due to the clipping in the loss function [3].

XII. Prototypes

In our initial priority queue implementation, we used multiple fixed-length queues for each priority level, requiring separate logic blocks for each priority. The updated architecture uses one continuous section of memory per priority queue (backed by BRAM), significantly saving on Configurable Logic Block usage. A fully pipelined design additionally overlaps 2-cycle MultiQueue read/write operations with 1-cycle CAM lookup and 1-cycle trade matching logic, enabling orders to be matched deterministically within 2 clock cycles.

XIII. Experiments and Results

1) **Decision Engine Performance**
   - Achieved a 4.314 ns critical path delay, operating stably at 200 MHz.
   - The 4-stage pipeline completes execution in 20 ns.
   - Evaluation with mock market data demonstrated 63% order capture accuracy.
2) **Trade Matching Engine Performance**
   - Achieves a throughput of 75 million orders per second using the novel MultiQueue architecture.
   - A fully pipelined design overlaps 2-cycle MultiQueue read/write with 1-cycle CAM lookup and 1-cycle trade matching logic.
   - Incoming orders are deterministically matched within 2 clock cycles, leveraging BRAM for predictable latency and a CAM controller for dynamic indexing.
3) **Software Infrastructure Performance**
   - Red-Black Tree insertion, search, and deletion verified correct via unit tests.
   - Average search time: $O(\log n)$, confirmed by benchmarking across 100,000+ orders.
   - Balancing verified correct based on black-height invariants and rotation tests.
4) **Reinforcement Learning / Neural Network Performance**
   - Backtesting accuracy: 63% on historical market data.
   - FPGA inference: one forward pass completed in 20 ns at 200 MHz.

References

[1] L. Kohútka, "Efficiency of Priority Queue Architectures in FPGA," *Journal of Low Power Electronics and Applications*, vol. 12, no. 3, p. 39, Jul. 2022, doi: https://doi.org/10.3390/jlpea12030039.

[2] M. Chandra, "On the implementation of fixed-point exponential function for machine learning and signal processing accelerators," *IEEE Design & Test*, vol. 39, no. 4, pp. 64–70, Aug. 2022, doi: https://doi.org/10.1109/mdat.2021.3133373.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA: MIT Press, 2018.