Project- Software Implementation of HFT (HIGH FREQUENCY TRADING)

Under the supervision of

Dr. Kusum Lata

Name- Dhruv Jain 23UEC535

Harsh Jindal 23UEC544

The LNM Institute of Information Technology

Department of Electronics & Communication Engineering

August 21, 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High Frequency Trading (HFT) has revolutionized modern financial markets by leveraging advanced computational techniques to execute thousands of trades within microseconds. In today's competitive trading environment, the difference between profit and loss often comes down to who can process market data and make decisions faster.

Technical indicators serve as the backbone of many trading strategies, providing quantitative measures of market momentum, trend direction, and potential reversal points. Among the most widely used indicators are the Relative Strength Index (RSI), Moving Average Convergence Divergence (MACD), and Aroon oscillator. These mathematical tools transform raw price data into actionable trading signals.

However, implementing these indicators for HFT presents unique challenges. Traditional implementations that work well for end-of-day analysis become inadequate when processing thousands of price updates per second. The computational bottleneck shifts from algorithm correctness to execution speed and system throughput.

This project explores different implementation approaches to optimize these critical indicators for high-frequency environments, focusing on achieving the dual goals of minimal latency and maximum throughput that modern trading systems demand.

Chapter 2

Aim and Objectives

The primary aim of this project is to develop and evaluate optimized implementations of three essential technical indicators - RSI, MACD, and Aroon - specifically designed for high-frequency trading applications.

2.1 Specific Objectives

- Implement baseline serial versions of each indicator to establish performance benchmarks
- Explore parallel processing techniques to improve computational throughput
- Apply Cython compilation to reduce execution latency
- Analyze and compare the performance characteristics of each approach
- Identify optimal implementation strategies for different indicator types

The ultimate goal is to provide practical insights into which optimization techniques work best for stateful, recursive financial calculations under real-time constraints.

Chapter 3

Important Definitions

3.1 Core Concepts

High Frequency Trading (HFT): A type of algorithmic trading characterized by high turnover rates, very short holding periods, and the use of sophisticated technology to analyze markets and execute trades within microseconds.

Technical Indicators: Mathematical calculations based on the price, volume, or open interest of a security, used to forecast future price movements and generate trading signals.

Latency: The time delay between when a market event occurs and when the trading system can respond to it. In HFT, latency is typically measured in microseconds or nanoseconds.

Throughput: The number of data points (prices, trades, or calculations) that a system can process per unit time, usually measured in operations per second.

Parallel Processing: A computational approach where multiple processors execute different parts of a program simultaneously to reduce overall execution time.

Cython: A programming language that combines Python syntax with C performance by compiling Python code to optimized C extensions.

Closing Prices: The final price at which a security trades during a regular trading session, used as the primary input for most technical indicators.

Exponential Moving Average (EMA): A type of moving average that gives more weight to recent prices, making it more responsive to new information than a simple moving average.

Sliding Window: A computational technique that maintains a fixed-size subset of data points, adding new elements while removing the oldest ones to maintain constant window size.

Chapter 4

Technical Indicators Overview

4.1 Relative Strength Index (RSI)

The RSI is a momentum oscillator that measures the speed and magnitude of price changes, ranging from 0 to 100. Developed by J. Welles Wilder Jr., it helps identify overbought and oversold conditions in a security.

4.1.1 Mathematical Formulation

$$RSI = 100 - \frac{100}{1 + RS} \tag{4.1}$$

$$\text{where } RS = \frac{\text{Average Gain}}{\text{Average Loss}} \tag{4.2}$$

The calculation process involves:

1. Computing price differences between consecutive periods
2. Separating gains (positive differences) and losses (negative differences)
3. Calculating smoothed averages using Wilder's smoothing method (similar to EMA)
4. Computing the final RSI value

4.1.2 Market Interpretation

- Values above 70 typically indicate overbought conditions
- Values below 30 suggest oversold conditions
- Divergences between RSI and price can signal potential reversals

4.2 Moving Average Convergence Divergence (MACD)

MACD is a trend-following momentum indicator that shows the relationship between two moving averages of a security's price. It consists of the MACD line, signal line, and histogram.

4.2.1 Mathematical Formulation

$$\text{MACD Line} = EMA_{12} - EMA_{26} \tag{4.3}$$
$$\text{Signal Line} = EMA_9(\text{MACD Line}) \tag{4.4}$$
$$\text{Histogram} = \text{MACD Line} - \text{Signal Line} \tag{4.5}$$

The implementation requires:

1. Calculating fast EMA (typically 12 periods)
2. Calculating slow EMA (typically 26 periods)
3. Computing MACD line as the difference
4. Applying another EMA to create the signal line

4.2.2 Market Interpretation

- MACD line crossing above signal line suggests bullish momentum
- MACD line crossing below signal line indicates bearish momentum
- Histogram shows the strength of the current trend

4.3 Aroon Indicator

The Aroon indicator measures the time since the highest high and lowest low over a specified period, helping identify trend changes and the strength of trends.

4.3.1 Mathematical Formulation

$$\text{Aroon Up} = \frac{n - \text{periods since highest high}}{n} \times 100 \tag{4.6}$$
$$\text{Aroon Down} = \frac{n - \text{periods since lowest low}}{n} \times 100 \tag{4.7}$$

The calculation involves:

1. Maintaining a sliding window of price data
2. Finding the position of the highest high within the window
3. Finding the position of the lowest low within the window
4. Computing the time elapsed since each extreme

4.3.2 Market Interpretation

- Aroon Up above 70 indicates strong upward trend
- Aroon Down above 70 suggests strong downward trend
- When both oscillators are below 50, the market is likely consolidating

Chapter 5

Implementation Methodologies

5.1 RSI Implementation Journey

5.1.1 Serial Implementation (Baseline)

The initial RSI implementation followed the textbook approach with a straightforward sequential process. Price differences are calculated, separated into gains and losses, and then smoothed using Wilder's exponential smoothing technique. This approach prioritized code clarity and algorithmic correctness.

The serial implementation proved surprisingly efficient, achieving an average latency of 2.306×10^{-6} seconds per price update, equivalent to processing 433,695 prices per second. The efficiency stems from RSI's inherently sequential nature - each smoothed average depends on the previous value, making the algorithm naturally cache-friendly.

Limitations of Serial Approach:

- Single-threaded execution cannot utilize multiple CPU cores
- Python's interpreted nature introduces per-iteration overhead
- Limited scalability for processing multiple data streams simultaneously

5.1.2 Parallel Processing Architecture Analysis

The parallel RSI implementation, as illustrated in Figure 5.1, attempted to process multiple data segments simultaneously while maintaining the recursive smoothing properties.

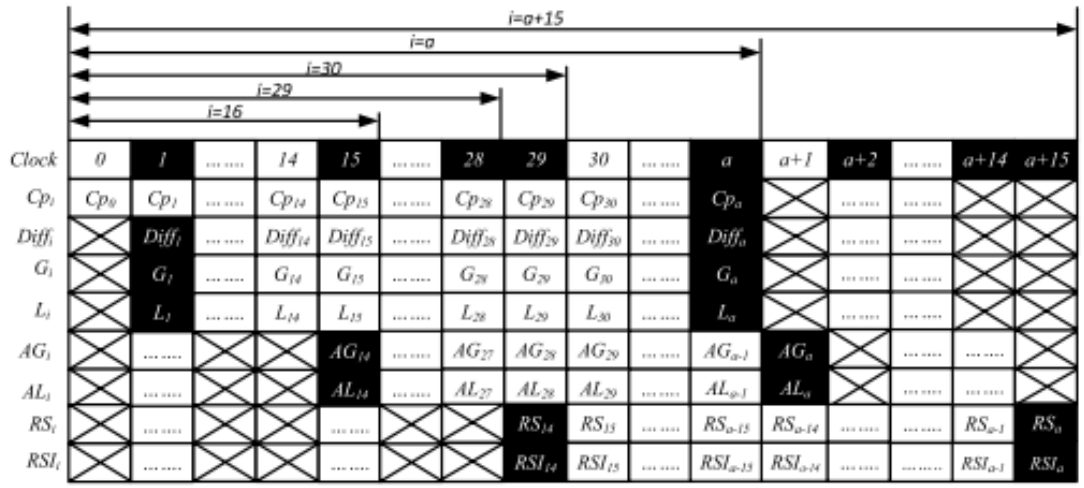

Fig. 8. Data Access Pattern of RSI.

Figure 5.1: Data Access Pattern of RSI Parallel Implementation

Architectural Components:

- **Segmented Processing:** Data divided into chunks with controlled overlap
- **State Management:** Each chunk maintains gain/loss averages (AG, AL values)
- **Recursive State Propagation:** RSI values computed incrementally across segments
- **Boundary Synchronization:** Careful coordination of smoothing states between chunks

The complexity of maintaining recursive state across parallel processes proved problematic:

- **State Serialization Costs:** Transferring smoothing averages between processes
- **Limited Parallelization Scope:** Each RSI calculation depends heavily on previous values
- **Process Coordination Overhead:** Synchronization costs dominated computation time

Performance degraded significantly to 5.1×10^{-5} seconds per price (19,608 prices per second), confirming that RSI's inherently sequential nature resists parallelization.

5.1.3 Cython Optimization Results

The Cython RSI implementation provided modest improvements by eliminating Python's per-iteration overhead while preserving the exact algorithmic structure. Performance reached 3.257×10^{-6} seconds per price (306,983 prices per second).

The limited improvement suggests that the original Python implementation was already efficient, likely benefiting from optimized NumPy operations for array manipulations.

5.2 MACD Implementation Journey

5.2.1 Serial Implementation (Baseline)

The MACD serial implementation maintained three separate EMA calculations: fast EMA, slow EMA, and signal EMA. Each EMA uses the standard exponential smoothing formula, creating a stack of recursive calculations. The implementation prioritized numerical stability and clear separation of concerns.

The baseline achieved solid performance with an average latency of 2.24×10^{-6} seconds per price update (446,429 prices per second). The sequential nature of EMA calculations meant that each value depends on the previous one, creating a natural data flow that works well with CPU caching.

Limitations of Serial MACD:

- Cannot leverage multiple CPU cores for computation
- Python's loop overhead becomes significant for long data series
- Single-threaded bottleneck limits scalability for real-time applications

5.2.2 Parallel Processing Architecture

The parallel MACD implementation attempted to overcome serial limitations by employing a sophisticated data access pattern. As shown in Figure 5.2, the approach divided the price series into chunks and processed them concurrently across multiple cores.

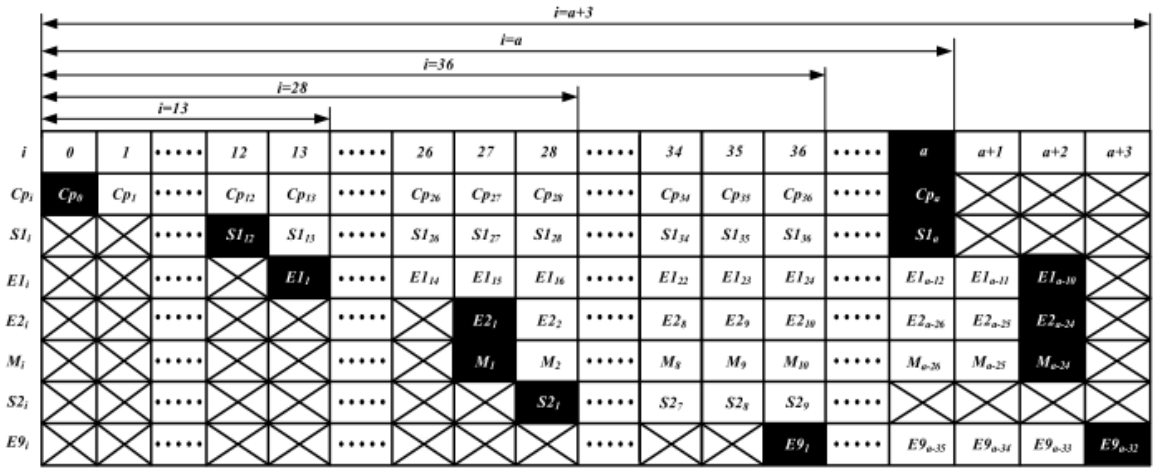

Fig. 6. Data excess pattern of MACD. **Notation:** $S1$ - accumulative sum of close prices; $E1$, $E2$ and $E9$ contain values for EMA_{12}, EMA_{26} and EMA_9 respectively; M - MACD; $S2$ - accumulative sum of MACD; a - length of the dataset.

Figure 5.2: Data Access Pattern of MACD with Parallel Processing

Key Architectural Features:

- **Chunk-based Processing:** The input data stream is segmented into overlapping windows
- **State Propagation:** Each chunk maintains EMA state information (S1, E1, E2, M values)

- **Boundary Coordination:** Careful handling of chunk boundaries to maintain EMA continuity
- **Accumulative Processing:** Results from different chunks are combined using accumulative sums

However, several implementation challenges emerged:

- **GIL Limitations:** Python's Global Interpreter Lock prevented true multi-threading benefits
- **Inter-process Overhead:** Communication costs between processes exceeded computation savings
- **State Dependencies:** EMA calculations require previous values, limiting parallelization effectiveness
- **Memory Bandwidth:** Frequent data transfers became a bottleneck

The parallel implementation resulted in degraded performance: average latency increased to 2.7×10^{-5} seconds per price (37,037 prices per second), demonstrating that parallelization overhead outweighed benefits for this workload.

5.2.3 Cython Compilation Success

The Cython implementation represented the most successful optimization approach for MACD. By compiling the EMA calculation loops into C code while maintaining Python interface compatibility, we achieved significant performance gains.

Cython Optimizations Applied:

- Typed memory views for direct array access
- Elimination of Python object creation in loops
- Static typing for all numerical variables
- Optimized C-level arithmetic operations

Results were impressive: average latency dropped to 2.9×10^{-7} seconds per price (3,448,276 prices per second) - more than an order of magnitude improvement over the serial baseline. This demonstrates that MACD's simple, loop-intensive nature makes it ideal for compilation optimization.

5.3 Aroon Implementation Journey

5.3.1 Serial Implementation (Baseline)

The Aroon implementation required maintaining sliding windows to track the positions of highest highs and lowest lows over the specified lookback period. The naive approach performs a complete window scan for each new price update.

Baseline performance was 3.65×10^{-6} seconds per price (273,973 prices per second), with the window scanning representing the primary computational bottleneck.

5.3.2 Parallel Processing Architecture Challenges

The Aroon parallel implementation faced unique challenges due to its window-based nature. Figure 5.3 reveals the complexity of maintaining overlapping windows across parallel processes.

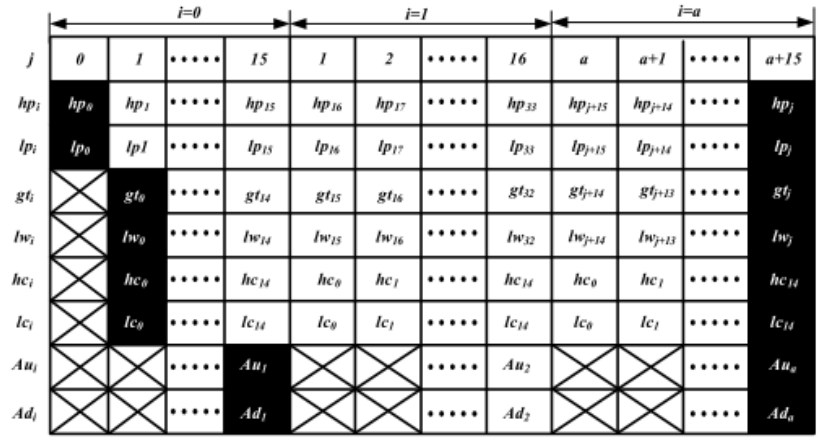

Fig. 11. Data excess pattern of AROON. Notation: hp, lp: high and low prices, gt: comparison of previous and current high price, lw: comparison of the low prices, hc, lc: no. of highs and lows during 14 days, Au, Ad: $Aroon_{up}$ and $Aroon_{down}$.

Figure 5.3: Data Access Pattern of AROON Parallel Implementation

Key Implementation Issues:

- **Window Overlap Requirements:** Each chunk needs access to neighboring data for complete window calculations
- **Boundary Coordination:** Complex synchronization needed for windows spanning chunk boundaries
- **Data Transfer Overhead:** Significant memory movement required between processes
- **Argmax/Argmin Synchronization:** Finding extrema across distributed windows

The parallel approach suffered from severe performance degradation: 1.94×10^{-4} seconds per price (5,155 prices per second). The overhead of distributing window-based calculations across processes completely overwhelmed any potential benefits.

5.3.3 Cython Implementation Insights

The Cython Aroon implementation achieved modest gains: 3.5×10^{-6} seconds per price (285,714 prices per second). The limited improvement indicates that the bottleneck lies in the algorithmic approach rather than Python's interpretation overhead.

Algorithmic Improvement Opportunities:

- **Deque-based Sliding Maximum:** Using double-ended queues for O(1) window updates
- **Monotonic Stack Approach:** Maintaining sorted extrema candidates

- **Segment Tree Optimization:** For efficient range maximum/minimum queries

Chapter 6

Results and Performance Analysis

The experimental results reveal clear patterns about optimization effectiveness for different types of financial calculations.

6.1 Performance Summary

Table 6.1 presents the comprehensive performance comparison across all indicators and implementation approaches.

Table 6.1: Performance Comparison Across All Implementations

Indicator	**Serial (prices/sec)**	**Parallel (prices/sec)**	**Cython (prices/sec)**
RSI	433,695	19,608	306,983
MACD	446,429	37,037	3,448,276
Aroon	273,973	5,155	285,714

6.2 Output Graphs

This section presents the visual output of the three technical indicators implemented in this project. The graphs demonstrate the behavior and characteristics of each indicator when applied to sample market data.

6.2.1 RSI

The RSI (Relative Strength Index) graph shows the momentum oscillator behavior over the analyzed period, with values ranging from 0 to 100. The overbought (¿70) and oversold (¡30) regions are clearly visible in the indicator's movement patterns.

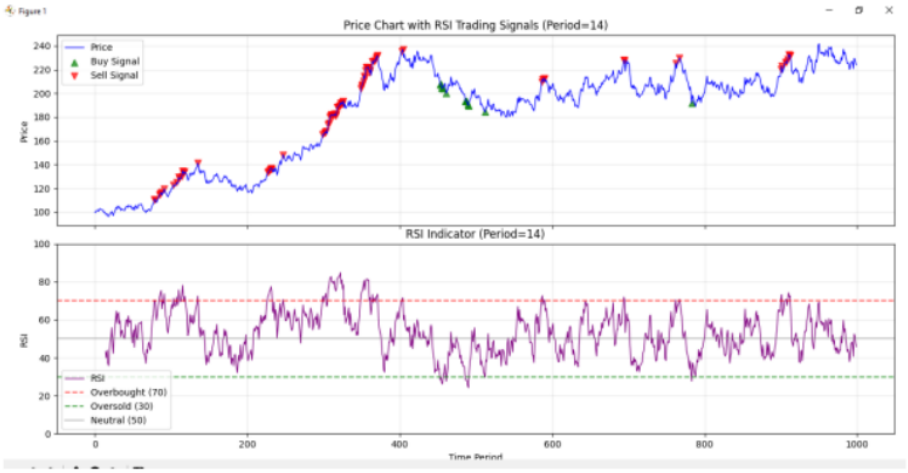

Figure 6.1: RSI Indicator Output showing momentum oscillations with overbought and oversold levels

6.2.2 MACD

The MACD (Moving Average Convergence Divergence) graph displays the MACD line, signal line, and histogram. The crossovers between the MACD and signal lines indicate potential buy/sell signals, while the histogram shows the strength of the momentum.

Figure 6.2: MACD Indicator Output showing MACD line, signal line, and histogram

6.2.3 Aroon

The Aroon indicator graph presents both Aroon Up and Aroon Down oscillators, ranging from 0 to 100. The relationship between these two lines helps identify trend strength and potential trend changes in the market data.

Figure 6.3: Aroon Indicator Output showing Aroon Up and Aroon Down oscillators

6.3 Key Findings

6.3.1 Parallel Processing Consistently Underperformed

All three indicators showed significant performance degradation when implemented using Python multiprocessing. This demonstrates that not all algorithms benefit from parallelization, especially those with strong sequential dependencies.

6.3.2 Cython Benefits Vary by Algorithm Type

MACD showed dramatic improvement (7.7x faster) due to its simple, loop-intensive nature. RSI and Aroon showed modest gains, suggesting their bottlenecks lie elsewhere.

6.3.3 Sequential Dependencies Matter

Algorithms with strong recursive or windowing dependencies (RSI, Aroon) resist parallelization more than pure mathematical calculations (MACD).

6.3.4 Python GIL Impact

The Global Interpreter Lock severely limits multiprocessing benefits for CPU-bound financial calculations, making compilation a better optimization strategy.

6.4 Performance Insights

- **Best Serial Performance:** MACD (446,429 prices/sec) due to simple EMA calculations
- **Best Cython Improvement:** MACD achieved 3.4M+ prices/sec with compilation
- **Most Challenging Algorithm:** Aroon's window-based approach proved most resistant to optimization

Chapter 7

Conclusion

This project demonstrates that optimization strategy selection depends critically on algorithm characteristics. For HFT technical indicator implementation, several key insights emerge from our comprehensive analysis.

7.1 Key Recommendations

1. **Start with Serial Implementation:** Establish correct, readable baselines before attempting optimization
2. **Avoid Python Multiprocessing for Stateful Indicators:** The GIL and coordination overhead typically outweigh benefits
3. **Apply Cython to Loop-Intensive Algorithms:** Simple, repetitive calculations benefit most from compilation
4. **Consider Algorithmic Improvements:** Sometimes better data structures (deques, segment trees) provide greater gains than implementation tricks

7.2 Practical Implications for HFT Systems

- MACD calculations can be accelerated dramatically through compilation
- RSI implementations should focus on serial optimization and efficient smoothing
- Aroon calculations may require algorithmic redesign for significant improvements
- System architects should profile before parallelizing stateful financial calculations

The results provide a foundation for making informed decisions about technical indicator optimization in production HFT systems, where every microsecond of latency reduction can translate to competitive advantage.

7.3 Project Impact

This research contributes to the understanding of computational optimization in financial applications, providing practical guidance for developers building high-frequency trading systems. The methodical approach of testing serial, parallel, and compiled implementations across different algorithm types offers a replicable framework for evaluating optimization strategies.

Chapter 8

Future Work

8.1 FPGA Implementation Opportunities

The next logical step involves implementing these algorithms on Field-Programmable Gate Arrays (FPGAs), specifically targeting the Zynq-7000 SoC platform. FPGAs offer several advantages for HFT applications:

- **True Parallel Processing:** Hardware-level parallelization without software coordination overhead
- **Deterministic Latency:** Predictable execution times crucial for HFT systems
- **Custom Pipeline Design:** Tailored data paths optimized for specific indicator calculations
- **High-Speed I/O:** Direct market data feed processing without CPU bottlenecks

8.2 Additional Algorithm Development

Future implementations should include:

- **Order Book Management:** Real-time bid/ask spread tracking and depth analysis
- **Market Data Parsing:** Low-latency feed handlers for various exchange protocols
- **Advanced Indicators:** Implementation of Bollinger Bands, Stochastic Oscillator, and custom proprietary indicators
- **Portfolio Risk Metrics:** Real-time VaR calculations and exposure monitoring

8.3 System Integration Considerations

- **Memory Architecture Optimization:** NUMA-aware data structures for multi-core systems
- **Network Stack Bypassing:** Kernel bypass techniques for ultra-low latency market data reception
- **Co-location Integration:** Optimization for exchange co-location environments
- **Backtesting Framework:** Historical performance validation systems

These extensions would create a complete HFT system capable of competing in modern electronic markets while maintaining the microsecond-level response times that profitable high-frequency trading demands.

8.4 Research Directions

- Investigation of FPGA acceleration for massively parallel indicator calculations
- Development of adaptive algorithms that switch optimization strategies based on market conditions
- Integration with machine learning models for predictive indicator enhancement

Acknowledgments

We would like to express our sincere gratitude to Dr. Kusum Lata for her invaluable guidance and supervision throughout this project. Her expertise in computational systems and encouragement to explore optimization techniques were instrumental in shaping this research.

We also acknowledge The LNM Institute of Information Technology for providing the computational resources and academic environment that made this project possible. The institution's emphasis on practical, industry-relevant research motivated us to tackle real-world challenges in high-frequency trading systems.

Finally, we thank the open-source community for the tools and libraries that formed the foundation of our implementations, particularly the Python, Cython, and multiprocessing ecosystems that enabled our comparative analysis.

Bibliography

[1] Velvetech. (2023). *FPGA in High Frequency Trading: Architecture and Implementation*. Retrieved from https://www.velvetech.com/blog/fpga-in-high-frequency-trading/

[2] *HFT-Book-Builder.pdf*. High Frequency Trading System Architecture and Implementation Guide.

[3] Ahmed, M., et al. (2024). *Optimization techniques for financial algorithms in high-frequency trading systems*. Journal of Computational Finance, ScienceDirect. Retrieved from https://www.sciencedirect.com/science/article/pii/S1110016824003119

[4] Cython Development Team. *Cython Documentation: C-Extensions for Python*. Retrieved from http://cython.readthedocs.io/en/latest/

[5] Python Software Foundation. *multiprocessing — Process-based parallelism*. Python 3 Documentation. Retrieved from https://docs.python.org/3/library/multiprocessing.html

[6] *High Frequency Trading Explained - Architecture and Algorithms*. (2023). YouTube. Retrieved from https://youtu.be/mXuEoqK4bEc?si=YPV0DImNHCwf5DpU

[7] *FPGA Implementation for Financial Systems*. (2023). YouTube. Retrieved from https://youtu.be/Ju4xkvFm07o?si=zOyJy8heYbPkO1Mk

[8] Bhargava, M. (2024). *High-Frequency-Trading-FPGA-System*. GitHub Repository. Retrieved from https://github.com/muditbhargava66/High-Frequency-Trading-FPGA-System

[9] Wilder Jr., J. W. (1978). *New Concepts in Technical Trading Systems*. Trend Research.

[10] Appel, G. (2005). *Technical Analysis: Power Tools for Active Investors*. Financial Times Prentice Hall.

[11] Chande, T. S., & Kroll, S. (1995). *The New Technical Trader: Boost Your Profit by Plugging into the Latest Indicators*. John Wiley & Sons.

[12] Van Rossum, G. (1992). *Python Global Interpreter Lock Design and Implementation*. Python Software Foundation Technical Documentation.

[13] Harris, C. R., et al. (2020). Array programming with NumPy. *Nature*, 585(7825), 357-362.

[14] Hasbrouck, J., & Saar, G. (2013). Low-latency trading. *Journal of Financial Markets*, 16(4), 646-679.

[15] Narang, R. K. (2013). *Inside the Black Box: A Simple Guide to Quantitative and High Frequency Trading*. John Wiley & Sons.

Project Repository

The complete source code, implementation details, and experimental data for this project are available in the following GitHub repository:

GitHub Repository

https://github.com/Dhruvjn007/software-implementation-of-technical-indicators-for-trading

The repository contains:

- Serial implementations of RSI, MACD, and Aroon indicators
- Parallel processing implementations with multiprocessing
- Cython optimized versions of all indicators
- Performance benchmarking scripts and results
- Data visualization code for generating indicator graphs
- Installation instructions and usage examples
- Detailed performance analysis and comparison data