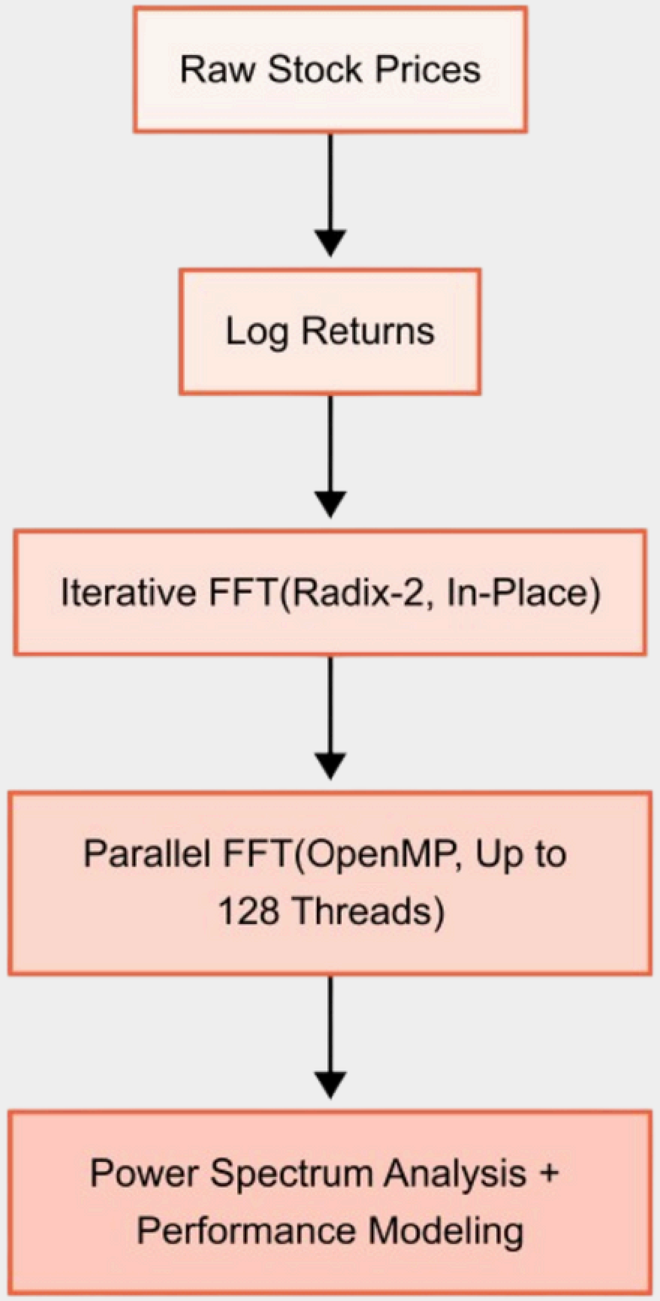
# High-Performance Iterative Fast Fourier Transform with Financial Application

ENG EC527 Spring 2025

PRESENTED BY

Krish Shah

# Problem & Motivation




**Problem Context**

The Fast Fourier Transform (FFT) is a core algorithm in high-performance computing and financial modeling. It converts time-series signals to the frequency domain with optimal complexity. However, traditional recursive FFTs are inefficient on modern processors due to poor cache locality and function call overhead, making them unsuitable for high-performance shared-memory systems.

**Project Motivation**

To address this, I implemented an iterative radix-2 Cooley-Tukey FFT in C and parallelized it using OpenMP, scaling to inputs of up to $2^{24}$ elements. The goal is to apply this high-performance FFT to real-world financial data like AAPL log returns, analyze the power spectrum, and validate both correctness and performance using thread scaling, memory modeling, and roofline analysis — all grounded in EC527 principles.

# Serial FFT Design

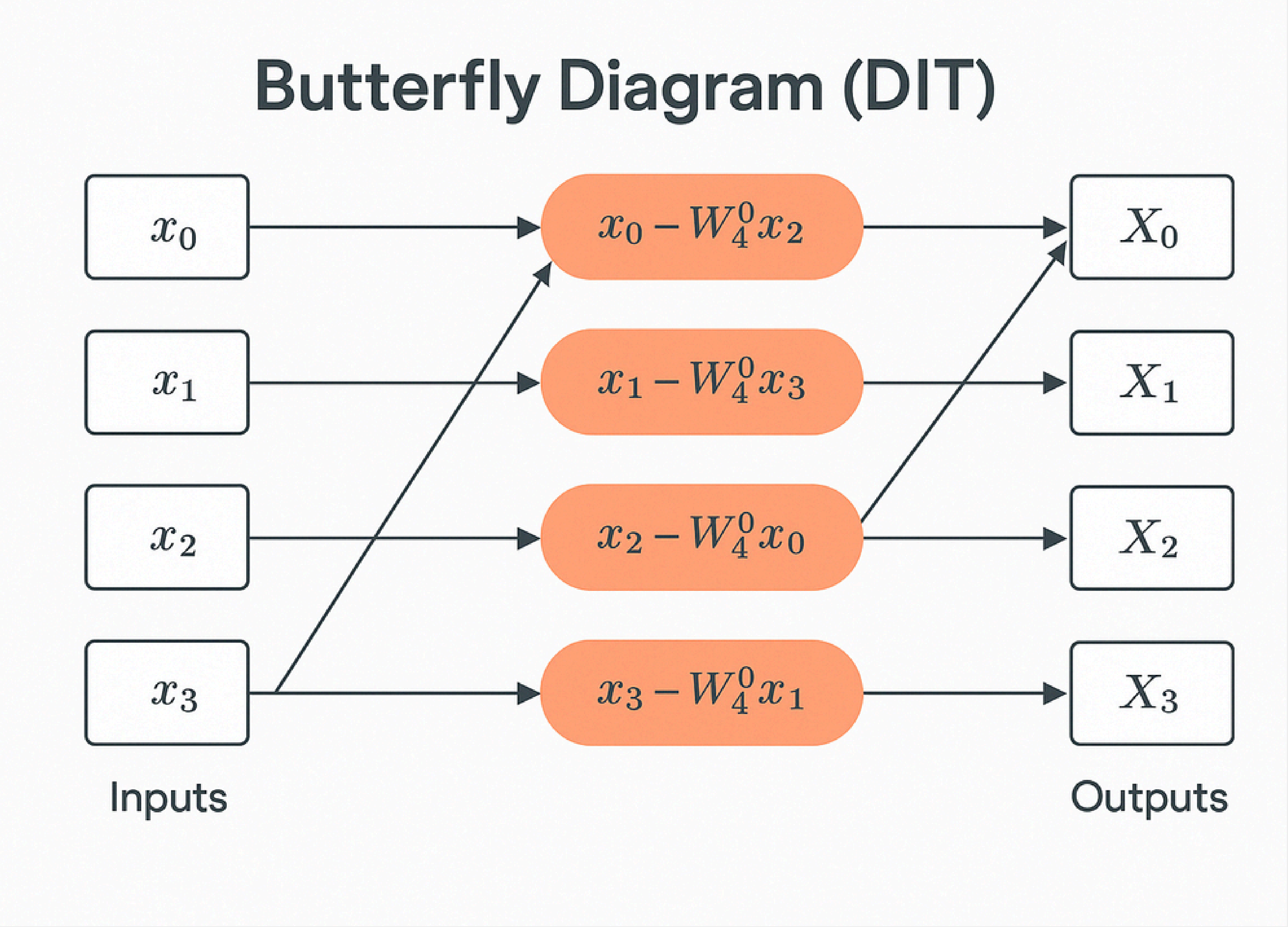
**Algorithm Structure**

The serial implementation is based on the radix-2 Cooley-Tukey algorithm, using an iterative decimation-in-time (DIT) strategy. It avoids recursion entirely by applying $\log_2 N$ stages of in-place butterfly operations, improving cache locality and enabling predictable memory access. Before each stage, the input array is reordered using bit-reversal permutation to ensure correct input ordering for the butterfly structure.

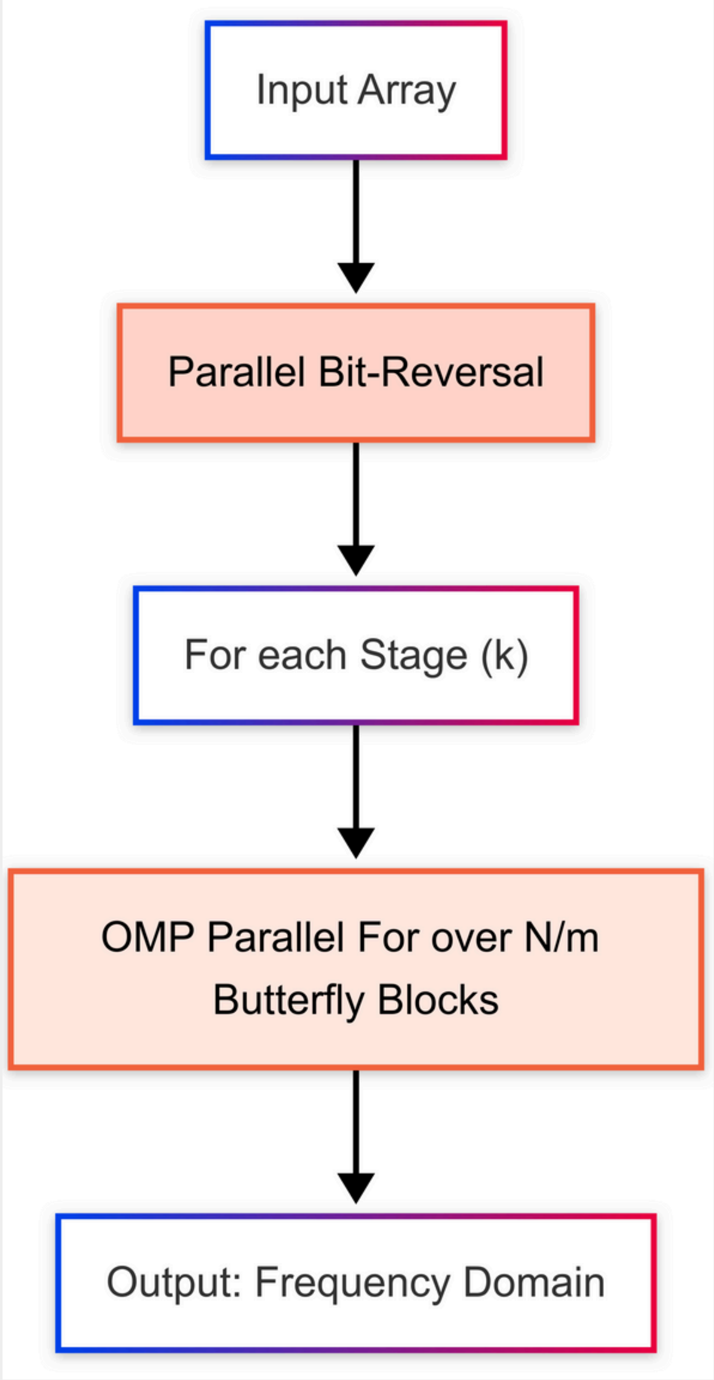
**Design Decisions**

The FFT operates on an array of complex double values, processed in-place to minimize memory usage. This approach enables tight control over memory layout and allows efficient use of CPU caches. By isolating bit-reversal from the main computation and avoiding unnecessary memory copies, the design provides a foundation for both correctness and performance. Results are validated against NumPy's FFT, with error under $10^{-10}$

$$X[k] = a + bW, \quad X[k+m/2] = a - bW$$



# Parallelization with OpenMP




**Targeted Parallelism**

To scale the FFT across multiple cores, we parallelized two independent parts of the computation using OpenMP:

- The bit-reversal permutation, which applies a deterministic reordering of elements
- The outer loop of each butterfly stage, where work is distributed across blocks of m-length computations

By parallelizing over independent butterfly blocks, we maximize core utilization while keeping memory accesses predictable and regular.

**Key Decisions & Observations**

We used #pragma omp parallel for schedule(static) to evenly distribute work and reduce scheduling overhead. False sharing was avoided by ensuring thread-local writes to distinct cache lines. At high thread counts, performance was limited by memory bandwidth, not computation — consistent with the low arithmetic intensity of FFT. Still, we achieved strong scaling up to 128 threads on large inputs.

# Optimization Techniques

**Design-Level Optimizations**

Beyond parallelism, multiple design-level optimizations were applied to reduce instruction count, improve memory locality, and avoid hardware-level bottlenecks. These include in-place memory updates, loop fusion, thread-private variables, and cache-aware scheduling.

**Why They Matter**

These optimizations collectively reduce cache misses, instruction overhead, and contention on shared memory resources. Combined with OpenMP, they enable the FFT to scale efficiently up to 128 threads on large inputs. Importantly, these are architecture-aware, aligning directly with EC527's emphasis on hardware-level performance reasoning.

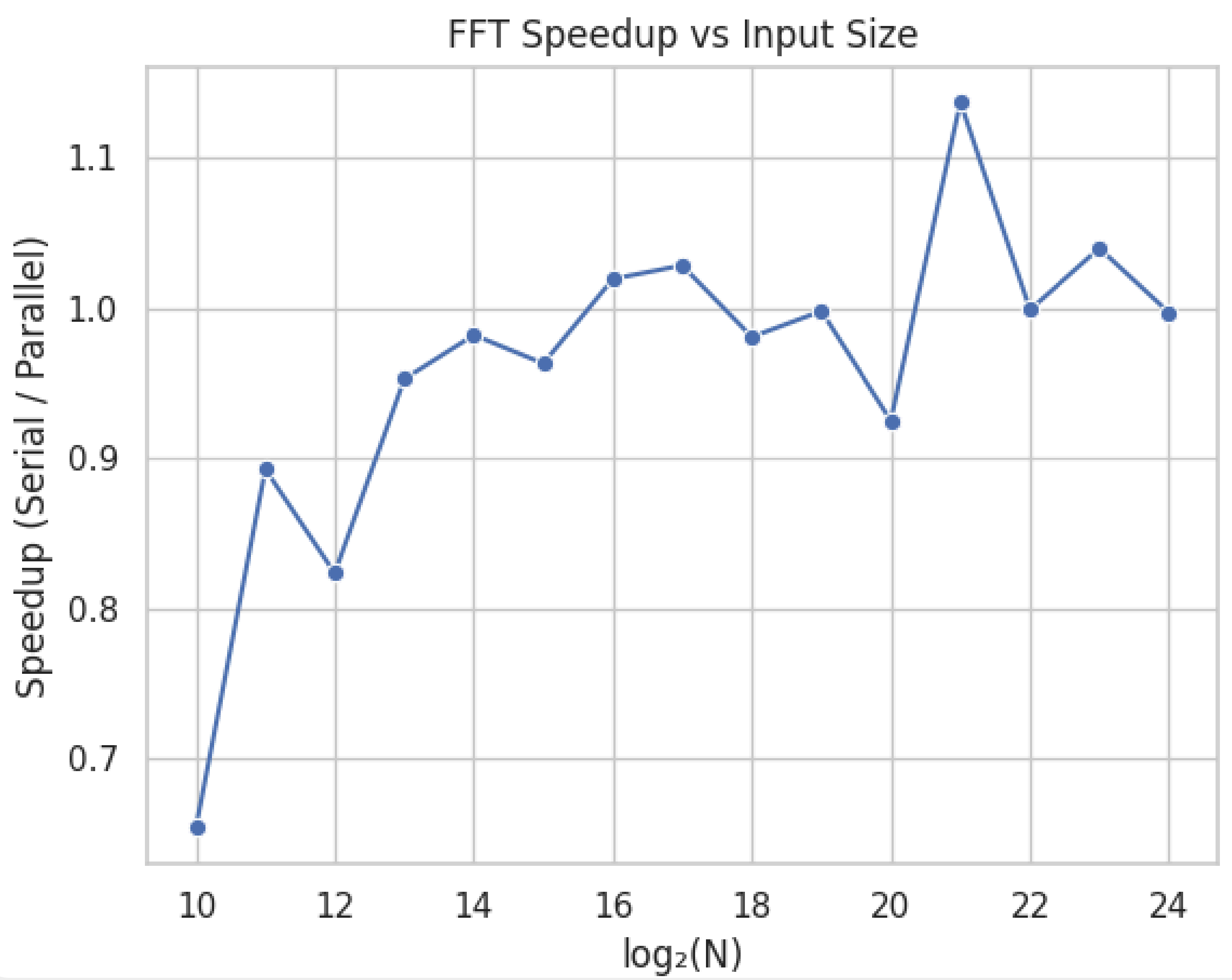
In-Place Memory

Loop Fusion

Thread-Private Buffers

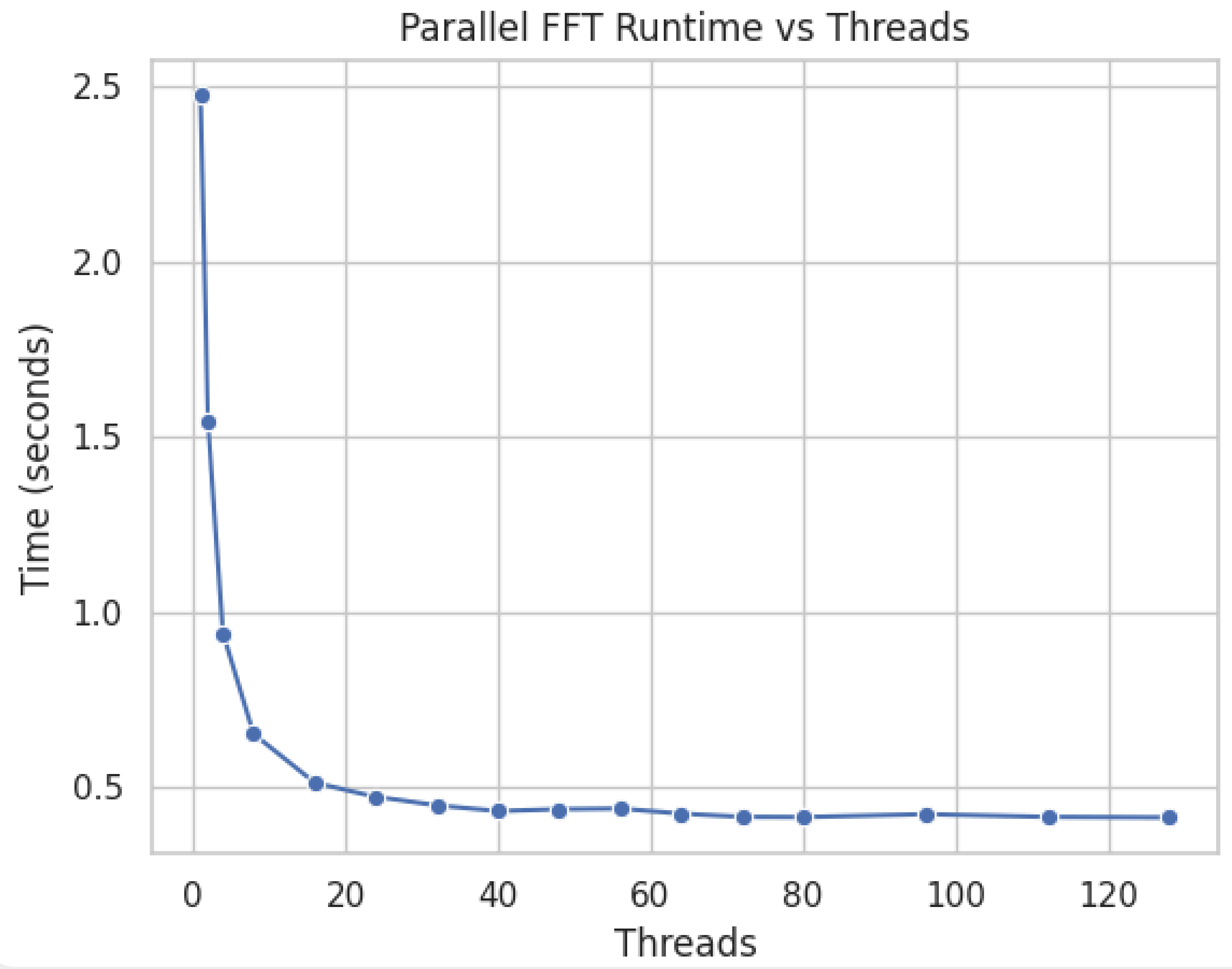
Static Scheduling

# Performance Results

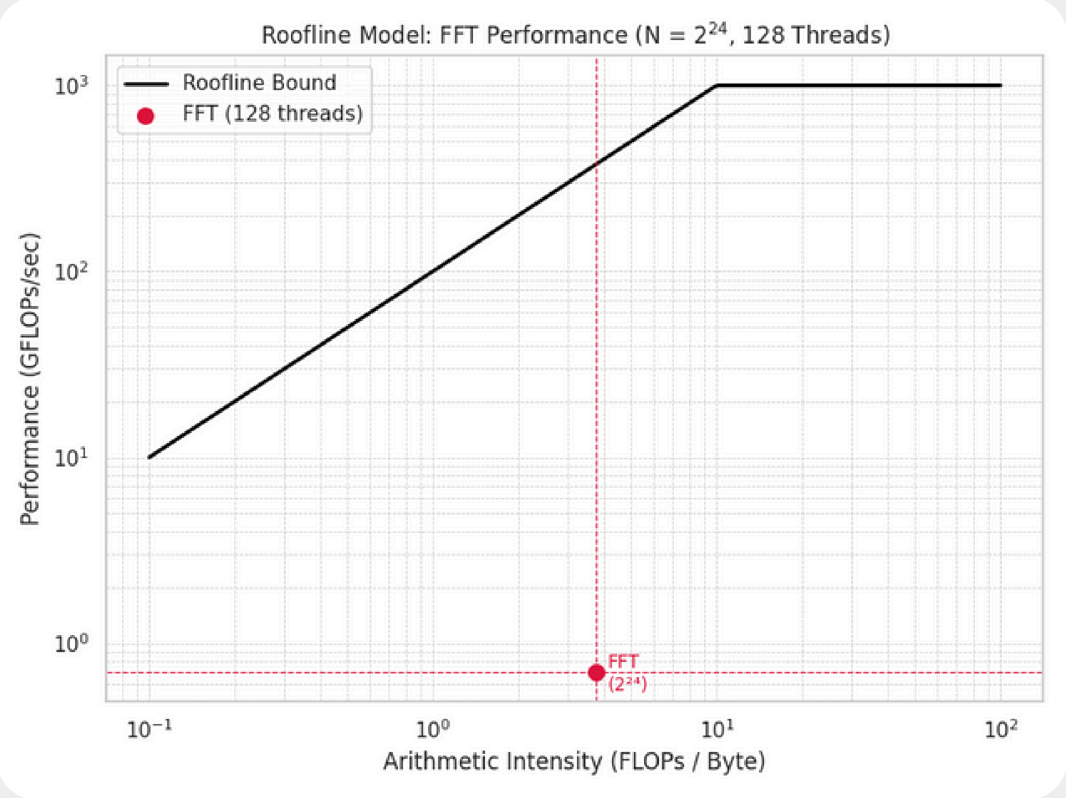


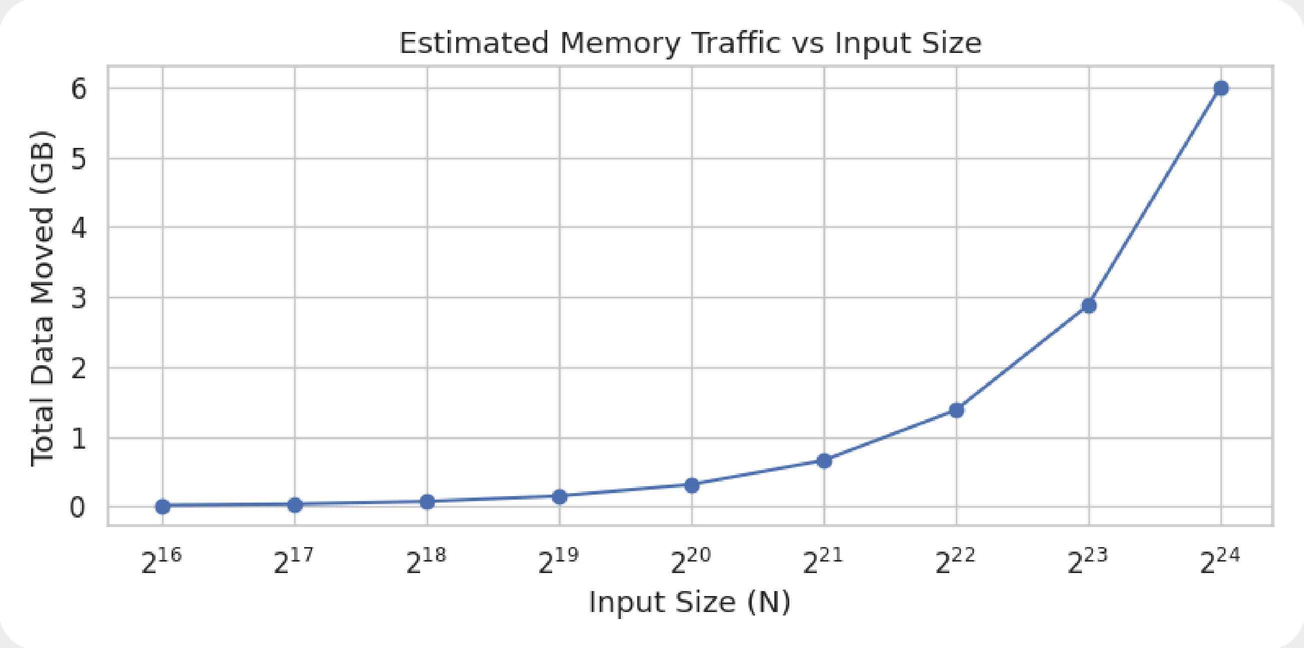

| N | Serial (s) | Parallel (s) | Speedup |
|---|---|---|---|
| $2^{14}$ | 0.00053 | 0.00055 | 0.95× |
| $2^{18}$ | 0.02254 | 0.02298 | 0.98× |
| $2^{20}$ | 0.09438 | 0.10200 | 0.93× |
| $2^{22}$ | 0.52157 | 0.52200 | 1.00× |
| $2^{24}$ | 2.47123 | 2.47782 | 1.00× |

# Thread Scaling & Efficiency




# Roofline Model & Hardware Counters



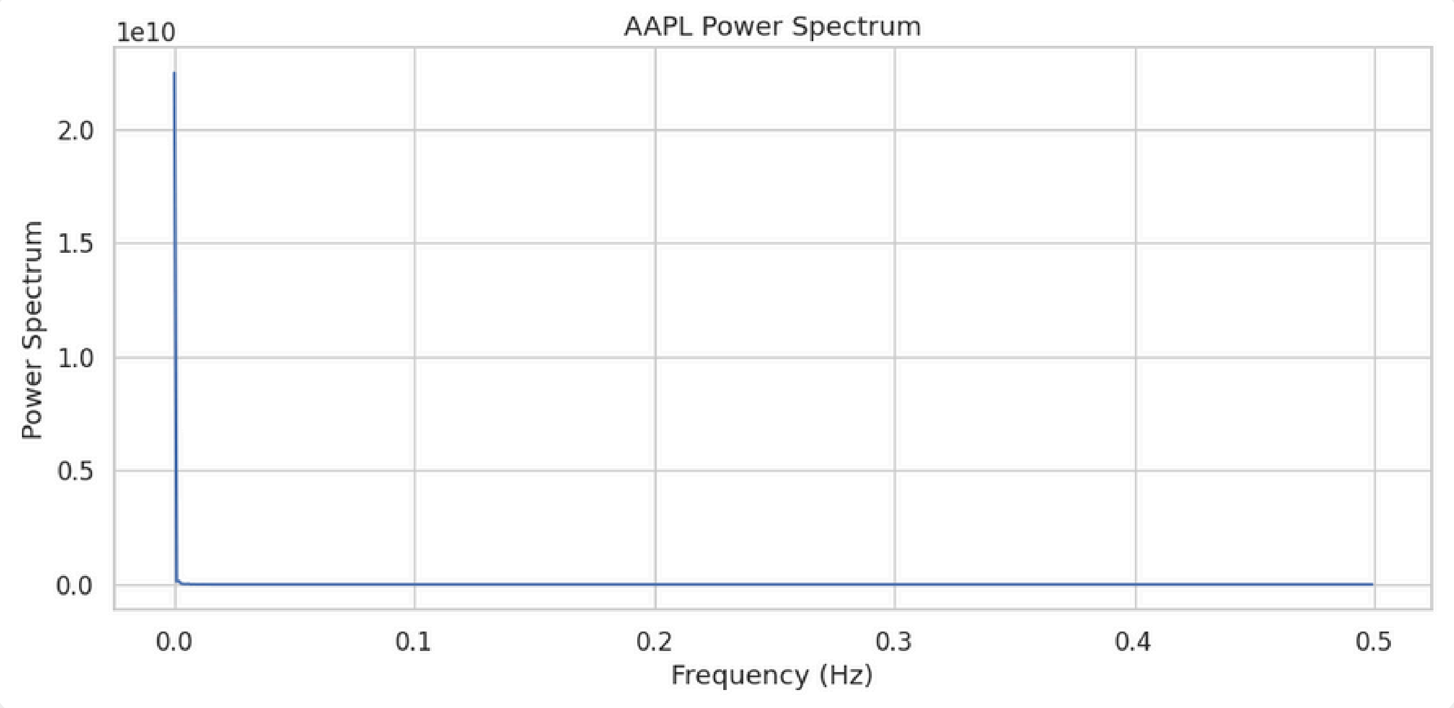

🧮 Roofline Metrics

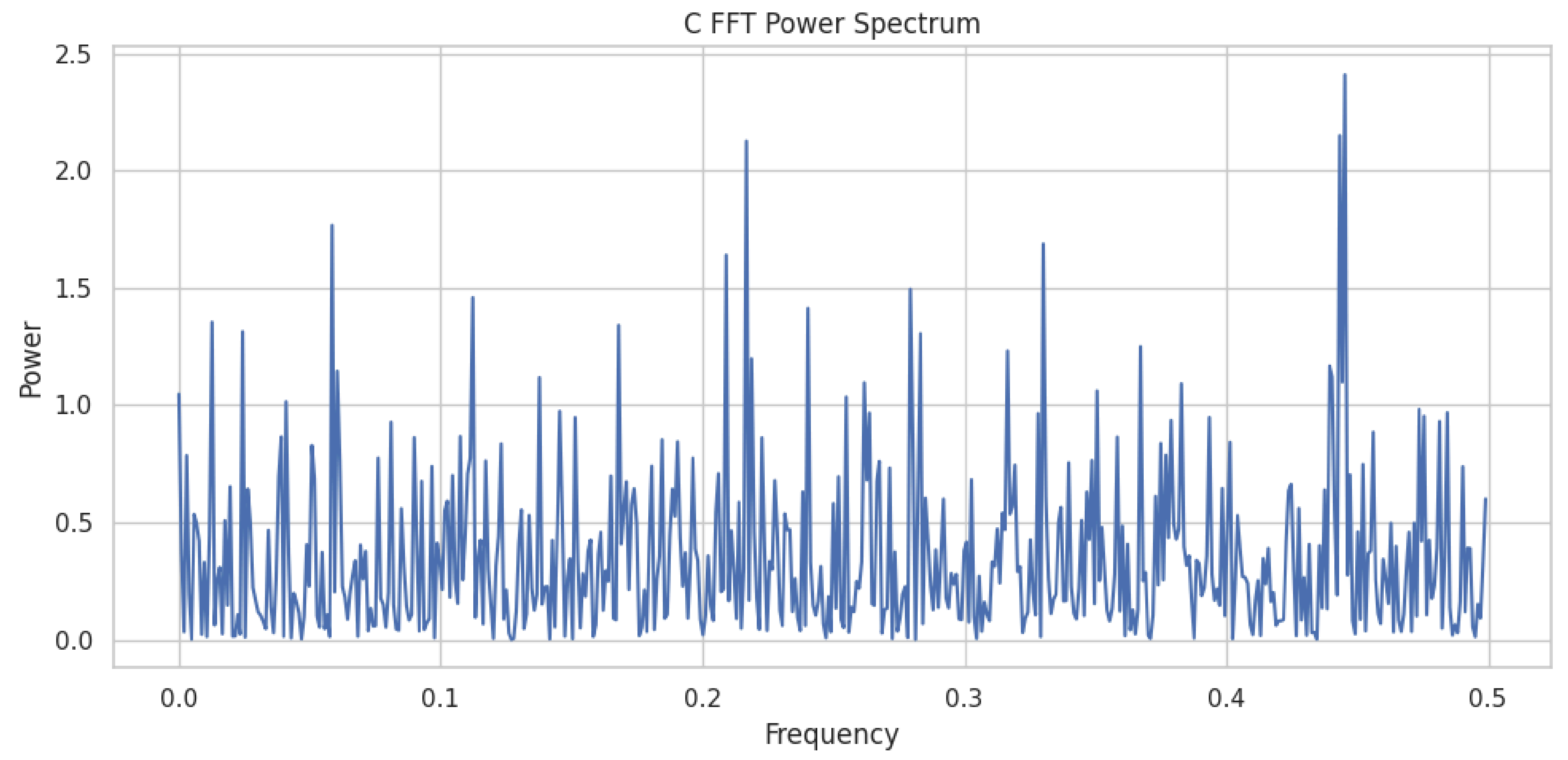
- Arithmetic Intensity: 2.74 FLOPs/Byte
- Measured Performance: 0.69 GFLOPs/sec
- Peak Bandwidth: ~100 GB/s
- Peak Compute: 1000 GFLOPs/sec
- **Conclusion: FFT lies far below both ceilings → memory-bound**

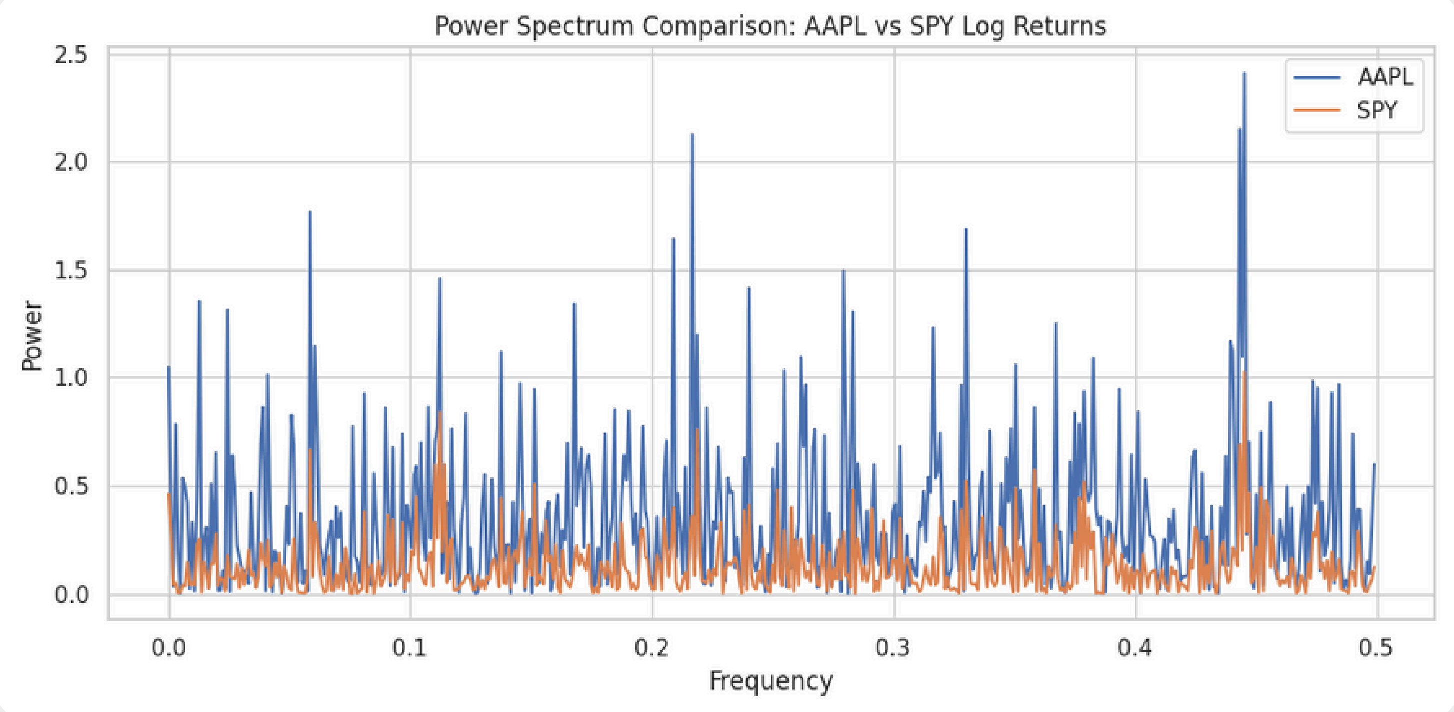
🧪 Hardware Counters (perf)

- Cache References: 131,279,761
- Cache Misses: 105,976,450
- Miss Rate: 80.7%
- Total Time: 12.44s
- FFT Time: 2.85s
- **Conclusion: High cache miss rate confirms bandwidth bottleneck**

# Financial Application: AAPL vs SPY




- The C FFT output matches NumPy's spectrum, validating correctness.
- AAPL shows more high-frequency content than SPY, indicating greater short-term volatility.
- FFT reveals structural differences in financial signals not visible in the time domain.

# Conclusions & Future Work

**Conclusions**

- Validated iterative FFT in C with OpenMP
- Scaled to $2^{24}$ inputs and 128 threads
- Roofline + perf confirm memory-bound behavior
- Applied to AAPL/SPY: revealed volatility in frequency domain

**Future Work**

- Add AVX2 SIMD to butterfly kernel
- Optimize thread placement (NUMA-aware)
- Extend to real-time/streaming financial data
- Integrate into ML workflows for market regime detection