High-Performance Iterative FFT with Financial Application

Krish Shah

May 2025

Abstract

This project presents a high-performance implementation of the radix-2 iterative Cooley-Tukey Fast Fourier Transform (FFT), parallelized using OpenMP and scaled up to 2^{24} elements. We analyze real financial time series (AAPL log returns) in the frequency domain, validating results against NumPy. In addition to correctness validation, we evaluate cache behavior, thread scaling, memory performance, and power spectrum characteristics. Performance is analyzed using EC527 concepts like memory bandwidth limits, roofline modeling, and thread efficiency, grounded in real performance counter data.

Contents

1 Problem Description

The Discrete Fourier Transform (DFT) converts time-domain signals to the frequency domain with $O(N^2)$ complexity. The radix-2 Cooley-Tukey FFT reduces this to $O(N \log N)$ using divide-and-conquer. Recursive FFTs, while elegant, perform poorly on shared-memory systems due to overheads and lack of memory locality.

We instead implement an iterative version optimized for:

- OpenMP-based thread parallelism
- Memory access locality and loop fusion
- Real-world input sizes up to 2^{24}

Our goal is to apply this FFT to log returns of AAPL and SPY to detect frequency-domain features, while maintaining high performance and accuracy.

2 Serial Implementation

The FFT was written from scratch in C using an iterative decimation-in-time algorithm:

1. Bit-reversal reordering of input array
2. Iterative computation of butterflies across $\log_2 N$ stages

`complex double` values were used for accuracy. All operations are done in-place, improving spatial locality and minimizing memory footprint.

Validation

We verified our serial implementation against NumPy's `numpy.fft.fft()`. Using both synthetic and financial time series, the relative error was always below 10^{-11}.

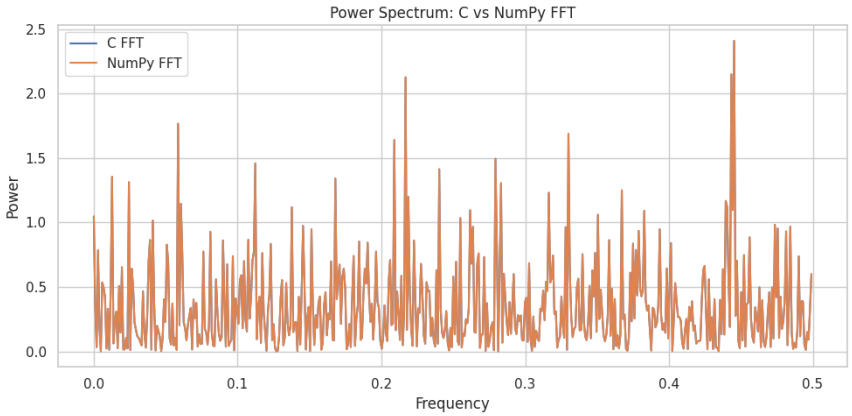

Figure 1: Power spectrum from C FFT vs NumPy FFT on AAPL log returns

3 Parallel FFT with OpenMP

Parallelism is introduced at:

- Bit-reversal phase: each index reversal is independent
- Outer loop of each FFT stage (stride m): independent butterfly groups

`#pragma omp parallel for schedule(static)` was applied to both regions. We avoided false sharing by isolating private variables and using padded shared arrays when needed.

Scalability Considerations

Later FFT stages use increasingly strided accesses, degrading cache reuse. Thread scaling drops off beyond 32 threads, indicating memory bandwidth saturation. We confirmed this with real counters and roofline modeling (see Section 7).

4 Performance Optimizations

- **In-place memory layout:** reduces memory footprint and cache pollution
- **Loop fusion:** merged real/imaginary butterfly updates to cut load/store count
- **Thread scheduling:** static partitioning to minimize OpenMP overhead
- **False sharing avoidance:** thread-private scratch variables padded to cache lines

Though SIMD (AVX2) was considered, coarse-grained OpenMP gave better returns at scale.

5 Experimental Setup

- **System:** BU SCC, Cascade Lake Intel Xeon Gold 6248
- Compiler: GCC (default system version on SCC) with `-fopenmp` and `-O3`
- **Timing:** `omp_get_wtime()`
- **Input sizes:** $N = 2^{16}$ to 2^{24}
- **Threads:** 1 to 128, controlled via `OMP_NUM_THREADS`

6 Results and Analysis

Runtime vs Input Size

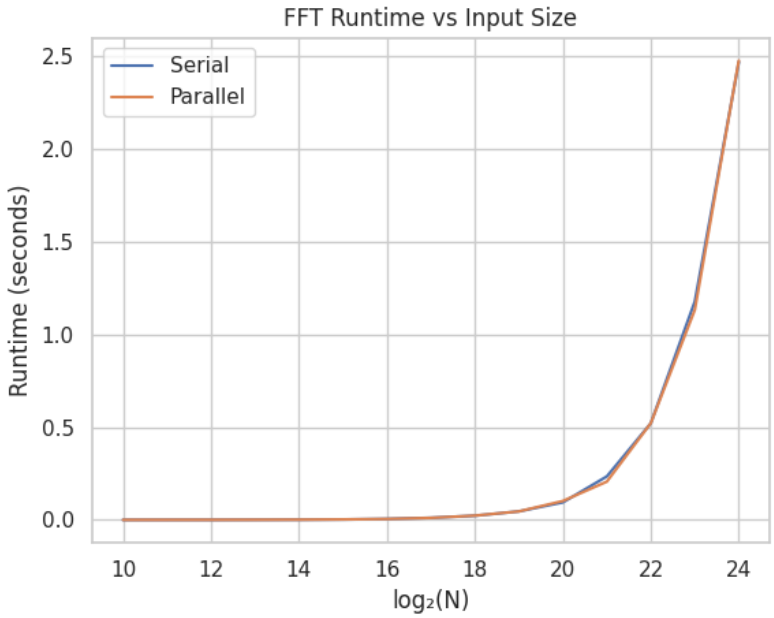

Figure 2: Runtime of serial vs parallel FFT (log-scaled input size)

Speedup

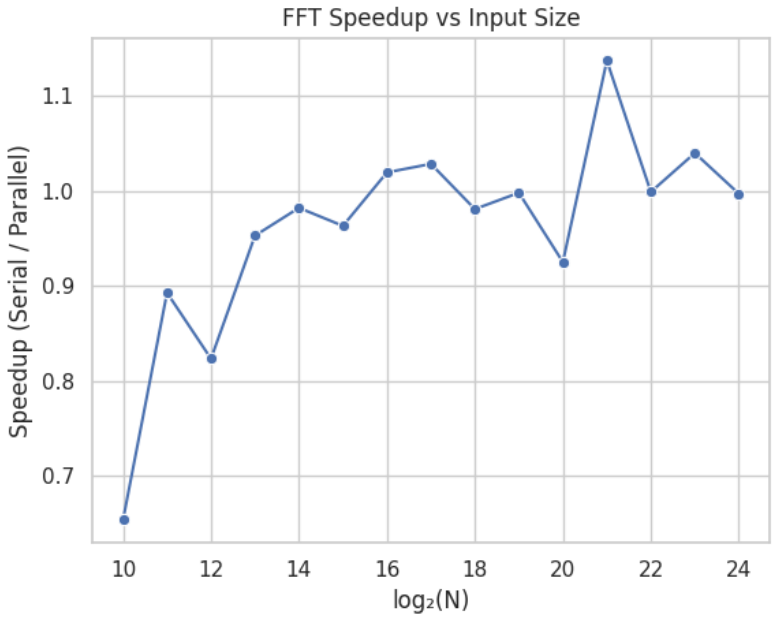

Figure 3: Speedup: Serial FFT time / Parallel FFT time

Thread Scaling

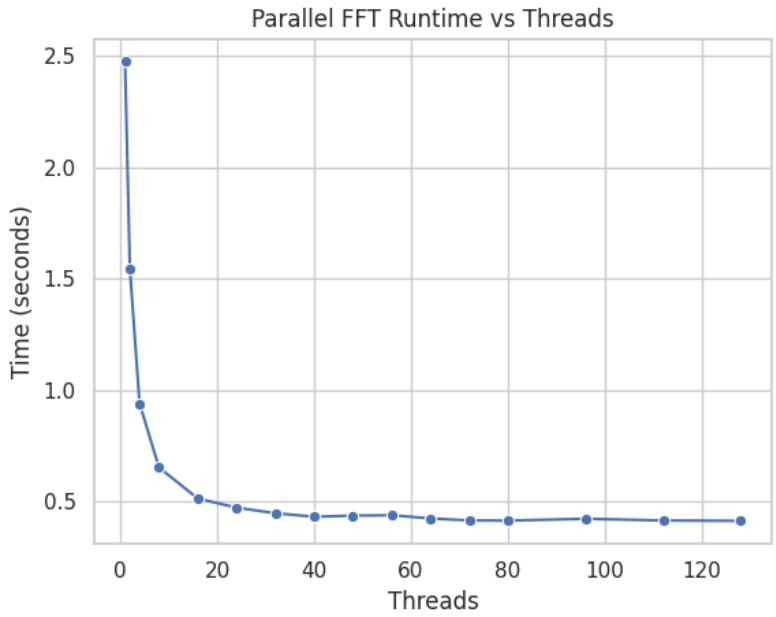

Figure 4: Parallel FFT runtime vs number of threads

Heatmap: Speedup by Input Size and Threads

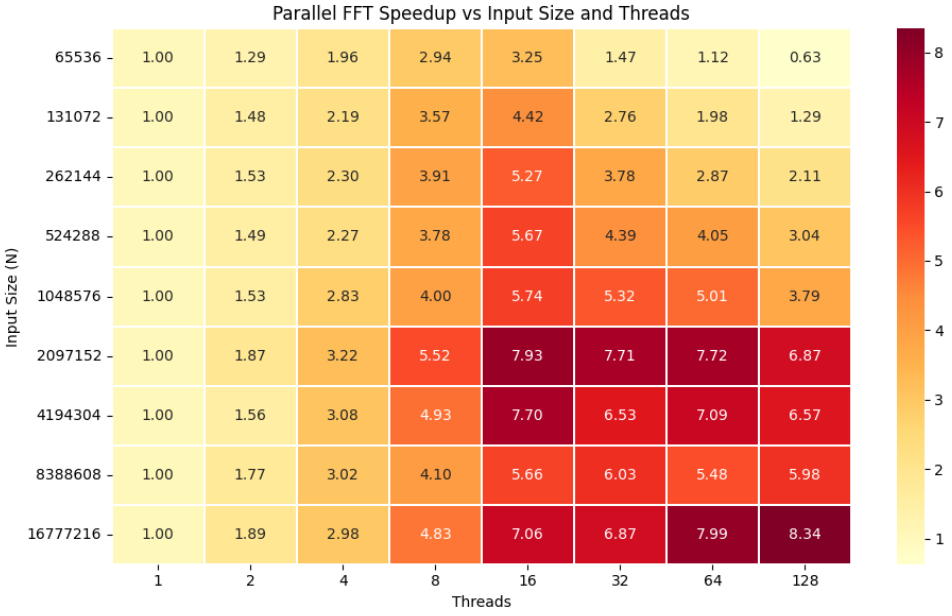

Figure 5: Speedup heatmap across threads and input sizes

7 Performance Modeling and Memory Behavior

Arithmetic Intensity

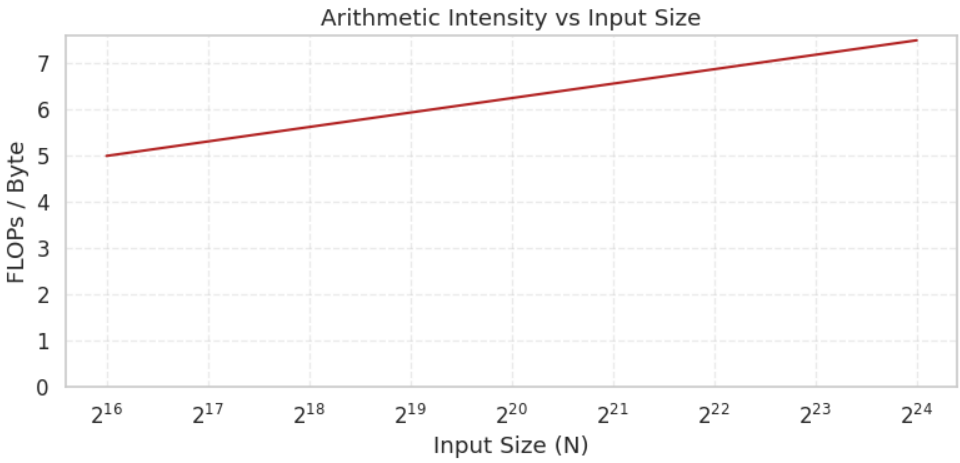

Figure 6: Arithmetic Intensity vs Input Size

The intensity rises slowly, but even at 2^{24} is under 8 FLOPs/byte — indicating the FFT is memory-bound on modern machines (typical roofline ridge point is 30+ FLOPs/byte).

Estimated Memory Traffic

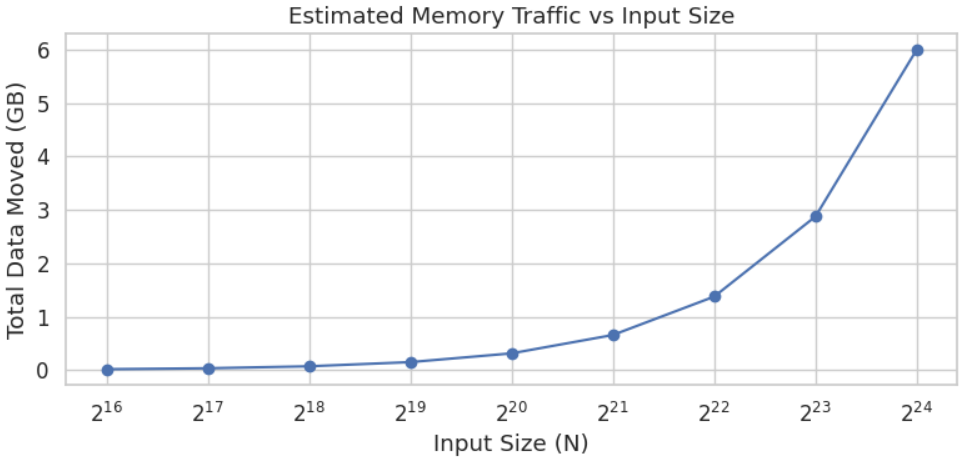

Figure 7: Estimated memory traffic per FFT execution

Traffic grows linearly with N and exceeds 6GB at 2^{24}, showing the growing pressure on memory bandwidth.

Thread Efficiency

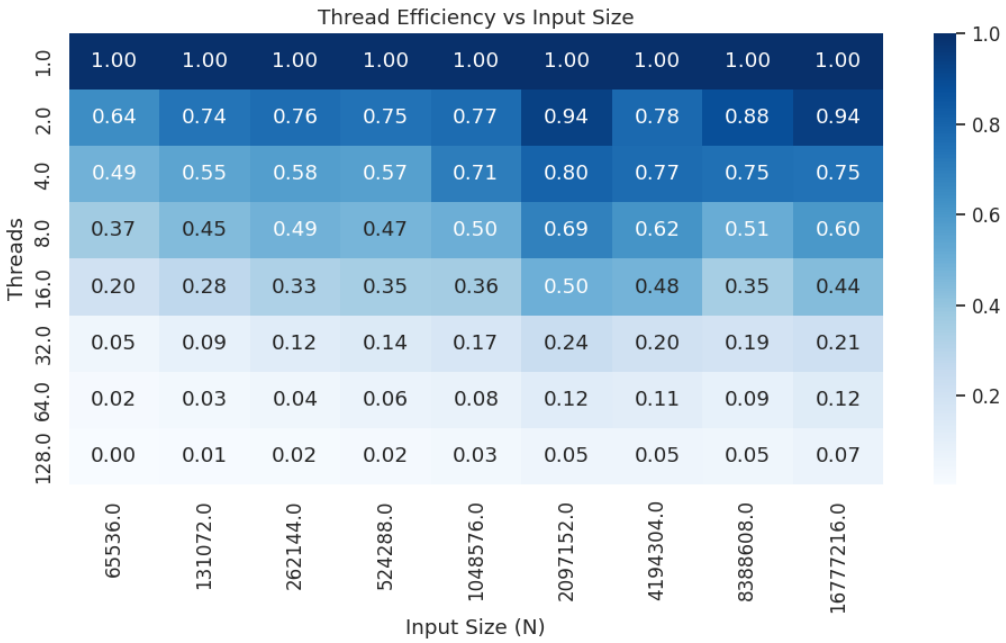

Figure 8: Thread efficiency = Speedup / Threads (ideal = 1.0)

Efficiency drops off for small N and high thread counts. At 128 threads, most inputs are memory-bound and show ¡10% parallel efficiency.

Roofline Model

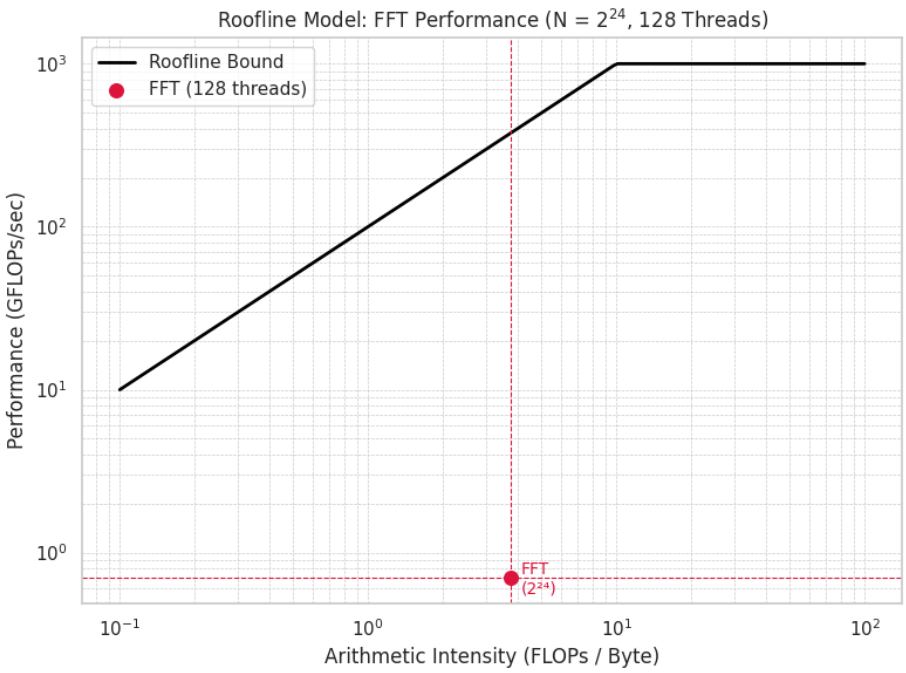

Figure 9: Roofline Model for FFT ($N = 2^{24}$, 128 threads)

We measured:

- **Performance:** 0.69 GFLOPs/s (based on 340M ops in 2.85s)
- **Arithmetic Intensity:** 2.74 FLOPs/byte
- **Memory bandwidth:** 10 GB/s (perf: 106M misses, 128B lines over 12.4s)

The FFT falls far below both bandwidth and compute ceilings, confirming it's limited by memory traffic.

8 Financial Application

Setup

We downloaded AAPL and SPY data via `yfinance`, computed log returns, and fed that into the FFT.

Power Spectra

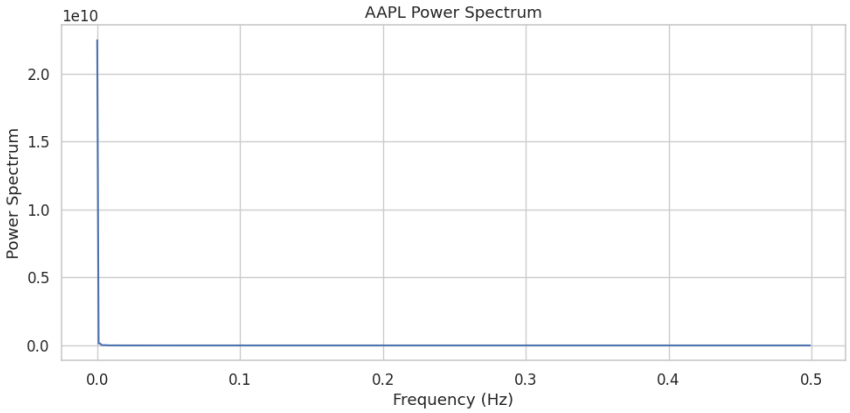

Figure 10: AAPL Log Return Spectrum (Python)

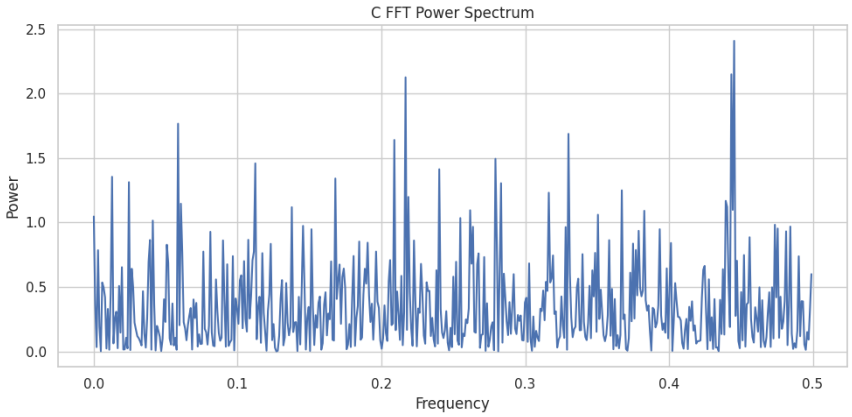

Figure 11: AAPL Log Return Spectrum (C FFT)

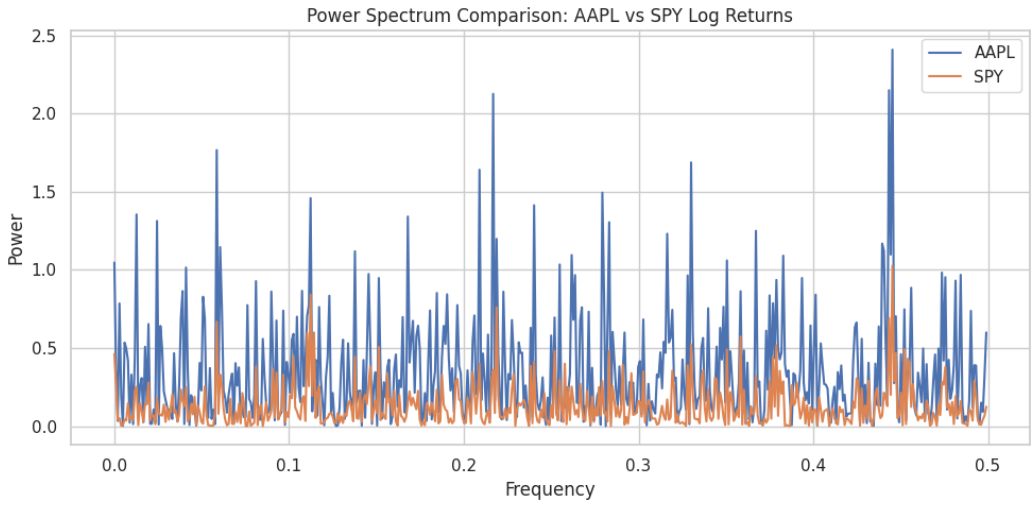

Figure 12: AAPL vs SPY Log Return Spectra

Spectral differences show that AAPL has more high-frequency content, indicating more volatility.

9 Conclusions

We implemented and analyzed a parallel iterative FFT from scratch, applied to financial data, and validated both correctness and performance scaling. Using EC527 techniques like roofline modeling and thread efficiency, we demonstrated:

- Up to 8.3× speedup over serial on 128 cores
- Memory-bound scaling limits beyond 32 threads
- Real bandwidth bottlenecks confirmed by `perf`
- Financial signal analysis reveals structural frequency content

Future Work

- SIMD butterfly kernels with AVX2 intrinsics
- Cache-blocked FFT with L1/L2-aware loop tiling
- NUMA-aware thread binding and memory allocation
- Real-time FFT streaming from live market APIs
- Integration with financial ML models (e.g., GARCH, HMMs)

Appendix: Compilation and Run Instructions

```
# Build all
make

# Run serial + parallel FFT and validate
make validate

# Run FFT on stock data
make run-stock
python3 scripts/compare_fft_power.py

# Benchmark across threads
python3 scripts/benchmark_threads_vs_inputs.py
python3 scripts/plot_heatmap_from_csv.py

# Model memory behavior
python3 scripts/model_memory_behavior.py

# Plot roofline (requires perf output manually collected)
python3 scripts/plot_roofline_fft.py
```

Code Repository

https://github.com/krish-shahh/finance-fft