Optimizing and Extending Periodogram Computations for Astronomy

A 300x faster and six orders of magnitude more accurate multi-term periodogram with finufft

Yuwei (Peter) Sun[1,2], Lehman Garrison[1]

[1] Scientific Computing Core (SCC), Flatiron Institute; [2] University of Illinois Urbana–Champaign

Introduction

The **Lomb–Scargle periodogram** is a standard tool in astronomy for detecting periodic signals in irregularly sampled time series, such as light curves from large-scale surveys.

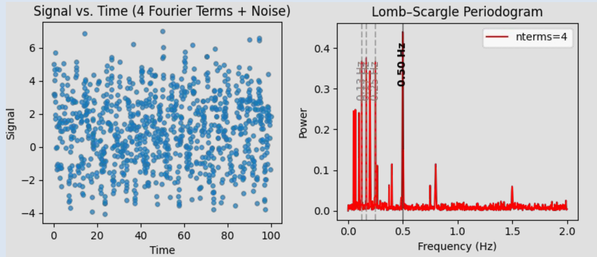

Figure 1: 4-Term Signal & Lomb–Scargle Periodogram

nifty-ls is a high-performance, more accurate implementation of the Lomb–Scargle periodogram, replacing the traditional sine and cosine summations with a type-1 NUFFT (via FINUFFT) that simultaneously yields both as the imaginary and real components of a single complex transform.

Challenge

Analyzing time-series data is fundamental to astronomy, but current tools force a difficult choice between speed and accuracy.

- **Accurate but Slow:** Astropy's *chi2* methods provide high accuracy but are computationally too expensive for large-scale astronomical data.
- **Fast but Inaccurate:** Faster FFT-based approximations(e.g. *fastchi2*) exist, but they sacrifice numerical accuracy.
- **The Parallelism Bottleneck:** Mainstream Python implement are single-threaded and cannot exploit multi-core CPUs or GPUs, leaving massive performance gains on the table and creating a critical bottleneck.

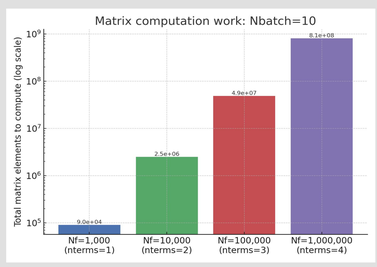

Figure 2: Impact of Nf and nterms on Computational Cost, where the size of matrix equals to $(1 + 2 \times nterms)^2$

Methods

This work firstly extended nifty-ls with multi-term harmonic fitting (Formula 1) and parallelized both pre- and post-processing with OpenMP, while fully leveraging **FINUFFT's** built-in parallelism to improve performance and scalability (Figure 3).

The **FINUFFT** (Flatiron Institute Nonuniform Fast Fourier Transform) is a high-performance library that runs the NUFFT on both CPUs and GPUs. Its advanced "exponential of semicircle" kernel enables a more compact and efficient computation, making it both significantly faster and more accurate than traditional methods.

$$M(t_j) = C_0 + \sum_{k=1}^{\text{nterms}} [A_k \cos(k\omega t_j) + B_k \sin(k\omega t_j)]$$

$$g_k = \sum_{j=1}^{N_d} h_j e^{ikt_j}$$

$$g_k = \sum h_j(\cos(kt_j) + i\sin(kt_j)) = \underbrace{\sum h_j \cos(kt_j)}_{C_k} + i \underbrace{\sum h_j \sin(kt_j)}_{S_k}$$

Formula 1: Lomb-Scargle Goal, NUFFT Definition, and Euler's Formula

1. Run a single NUFFT on the non-uniform data to get a complex sum.
2. Apply Euler's formula to separate the sum into its real and imaginary parts.
3. The real part becomes the cosine sum (Ck) and the imaginary part becomes the sine sum (Sk).

nterms: number of Fourier Terms; **gk**: NUFFT sum; **tj**: unevenly spaced observation times point j; **ck**: cosine sum; **sk**: sine sum

Profiler data shows Matrix Build and LU Solve as bottlenecks. So:

- CPU-based Optimization:
  - C++ with **nanobind** to reduce Python interpreter overhead and runtime latency.
  - **OpenMP static scheduling** to efficiently parallelize a large number of small, independent tasks (Figure 4).
  - **Customized LU solver** and **matrix multiplication routines** to avoid thread oversubscription caused by BLAS's internal parallelism.
- GPU Acceleration:
  - **CuPy** offloads computation to the GPU, accelerated for large frequency grids.

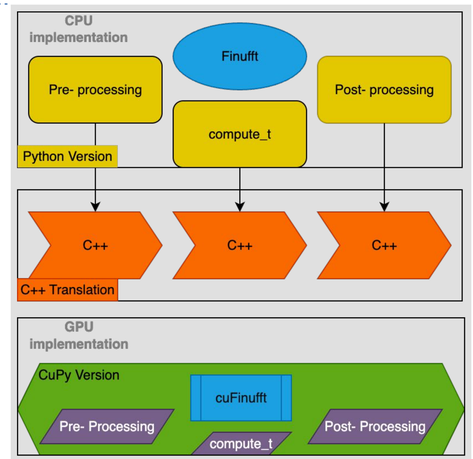

Figure 3: Hybrid CPU–GPU Acceleration Workflow in nifty-ls

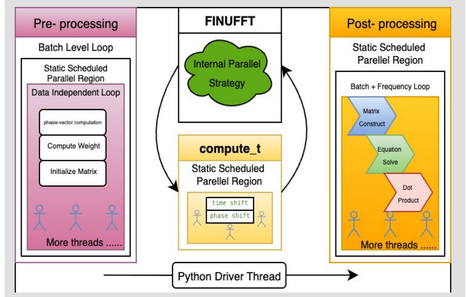

Figure 4: Single series Parallel Execution Strategy

Single-Series Mode

- **Input:** An unevenly spaced time series of N floats (with optional weights), representing observation values at specific timestamps.
- **Output:** A uniform periodogram of M floats, representing spectral power at M evenly spaced frequencies.

Heterobatch Mode

- **Input:** B independent unevenly spaced time series, each with its own length N_i and optional weights.
- **Output:** B uniform periodograms, each of length M.

Heterobatch method (Figure 5) is designed to efficiently process a large number of small, independent time series. Its input/output design exposes new parallelism opportunity. A Python ThreadPoolExecutor approach was limited by the GIL, so we switched to a C++ backend with **OpenMP Dynamic scheduling**, balancing variable-length series and minimizing overhead for efficient CPU use.

Scan for details about nifty-ls

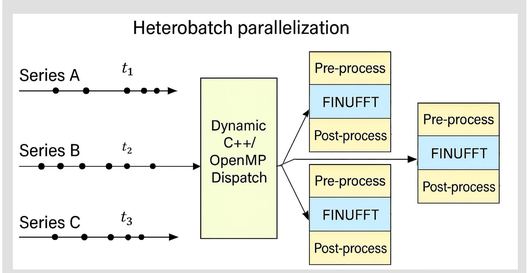

Figure 5: Heterobatch parallelization workflow

Conclusion

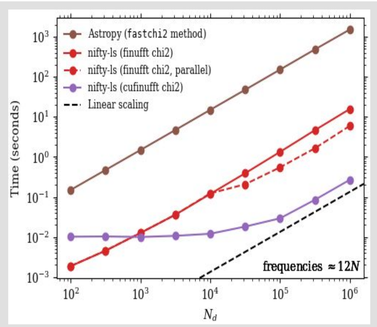

Figure 6: Performance Benchmark

On an **Intel Ice Lake CPU**, our implementation achieved ***100×*** speedup over Astropy's *fastchi2* in single-core tests, and ***300×*** speedup using **16 cores**. On an **NVIDIA A100 80GB GPU**, we observed an additional ***5600×*** acceleration.

For accuracy, both our CPU and GPU implementations were compared against Astropy's fastchi2 (*FFT-approximation* and *trigonometric* method). Achieve errors around 10^{-9}, offering a conservative **six orders of magnitude improvement** in accuracy.

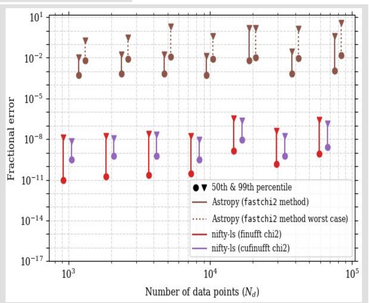

Figure 7: Accuracy Comparison

Name (time in ms)	Min	Mean	StdDev
test_batched_standard[cufinufft]	5.6997 (1.0)	6.1075 (1.0)	0.4922 (10.87)
test_heterobatch_standard[finufft_heterobatch]	9.1420 (1.60)	9.6008 (1.57	0.2600 (5.74)
test_batched_standard[finufft]	15.9838 (2.80)	24.1990 (3.96)	4.7521 (105.00)
test_unbatched_standard[finufft]	131.9397 (23.15)	132.2553 (21.65)	0.1672 (3.69)
test_unbatched_standard[astropy]	153.7311 (26.97)	153.8238 (25.19)	0.0453 (1.0)
test_unbatched_standard[cufinufft]	265.6694 (46.61)	269.3043 (44.09)	2.4806 (54.81)

Sheet 1: Performance Comparison between Hetrobatch, Batch and Unbatch

Using the same hardware and identical series count/size, the Heterobatch version delivered **>40%** performance improvement over the batch implementation.

Future Work

1. **GPU Heterobatch Support**: Implement heterobatch GPU version using Python 3.14 (no GIL) and CUDA Streams for efficient multi-term processing.
2. **Distributed Parallelism**: Split workloads across multiple GPUs to avoid out-of-memory issues with large frequency grids and multiple batches.

References:

1. Flatiron Institute. nifty-ls: Nonuniform Iterative Fast χ^2 Lomb–Scargle. GitHub repository. https://github.com/flatironinstitute/nifty-ls (2025)
2. VanderPlas, J. T. Understanding the Lomb–Scargle Periodogram. ApJS 236.1:16 (2018)
3. Flatiron Institute. FINUFFT: Flatiron Institute Nonuniform Fast Fourier Transform. Online documentation. https://finufft.readthedocs.io/en/latest/ (2025)
4. Astropy Collaboration. LombScargle — Astropy Documentation. Online documentation. https://docs.astropy.org/en/latest/timeseries/lombscargle.html (2025)