

Dedispersion Algorithm Implementation Comparison

This document is a high-level comparison of various implementations of dedispersion algorithms.

This work evaluated three types of dedispersion algorithms implemented in various programming languages (Python, Julia, and CUDA):

Brute-Force

This algorithm sums along each trial dispersion curve for each time sample. This results in many redundant calculations as it is the unoptimized but straightforward implementation of dedispersion.

Theoretical Algorithmic Complexity: $n_freq * n_time * n_trials$

Fast Dispersion Measure Transform (FDMT)

This algorithm reduces redundant computations by recursively processing two sub-bands of the data.

Algorithmic complexity: $\max\{n_time * n_trials * \log_2(n_freq), 2 * n_freq * n_time\}$

Original paper: <https://arxiv.org/pdf/1411.5373.pdf>

Taylor-Tree

This algorithm regularizes the problem to then exploit redundant computation between trial DMs.

Algorithmic complexity: $n_time * n_freq * \log_2(n_freq)$

Specific Implementations Evaluated:

CPU FDMT in Python (pyfdmt)

Codebase: <https://bitbucket.org/vmorello/pyfdmt/src/master/>

CPU FDMT in Julia (FastDMTransform.jl)

Codebase: <https://github.com/max-Hawkins/FastDMTransform.jl> Note: This fork is used to test against a single-threaded FDMT implementation that's a more fair comparison to pyfdmt.

GPU FDMT in Python (from Bifrost)

Codebase: <https://github.com/ledatelescope/bifrost>

CPU Brute-Force Dedisperion in Julia (Dedisp.jl)

Codebase: <https://github.com/max-Hawkins/Dedisp.jl> Note: This fork is used to remove the loop vectorization helper to make a more fair comparison. Kiran Shila is the primary author of Dedisp.jl.

GPU Brute-Force Dedisperion in Julia (Dedisp.jl)

Codebase: <https://github.com/max-Hawkins/Dedisp.jl> Note: This fork is used to remove the normalization step after dedisperion. Kiran Shila is the primary author of Dedisp.jl.

Why??? - How is this relevant to Breakthrough Listen or SETI technosignature searches?

These dedisperions are very much of interest to Breakthrough Listen. With minor tweaks to the algorithms, **all of these implementations can be applied to linear-drifting signals**. FDMT is especially easy to reimplement because it takes a parameter for the frequency-dependent time delay. The others are somewhat harder, but in theory, all of these algorithms are directly applicable to doppler searches.

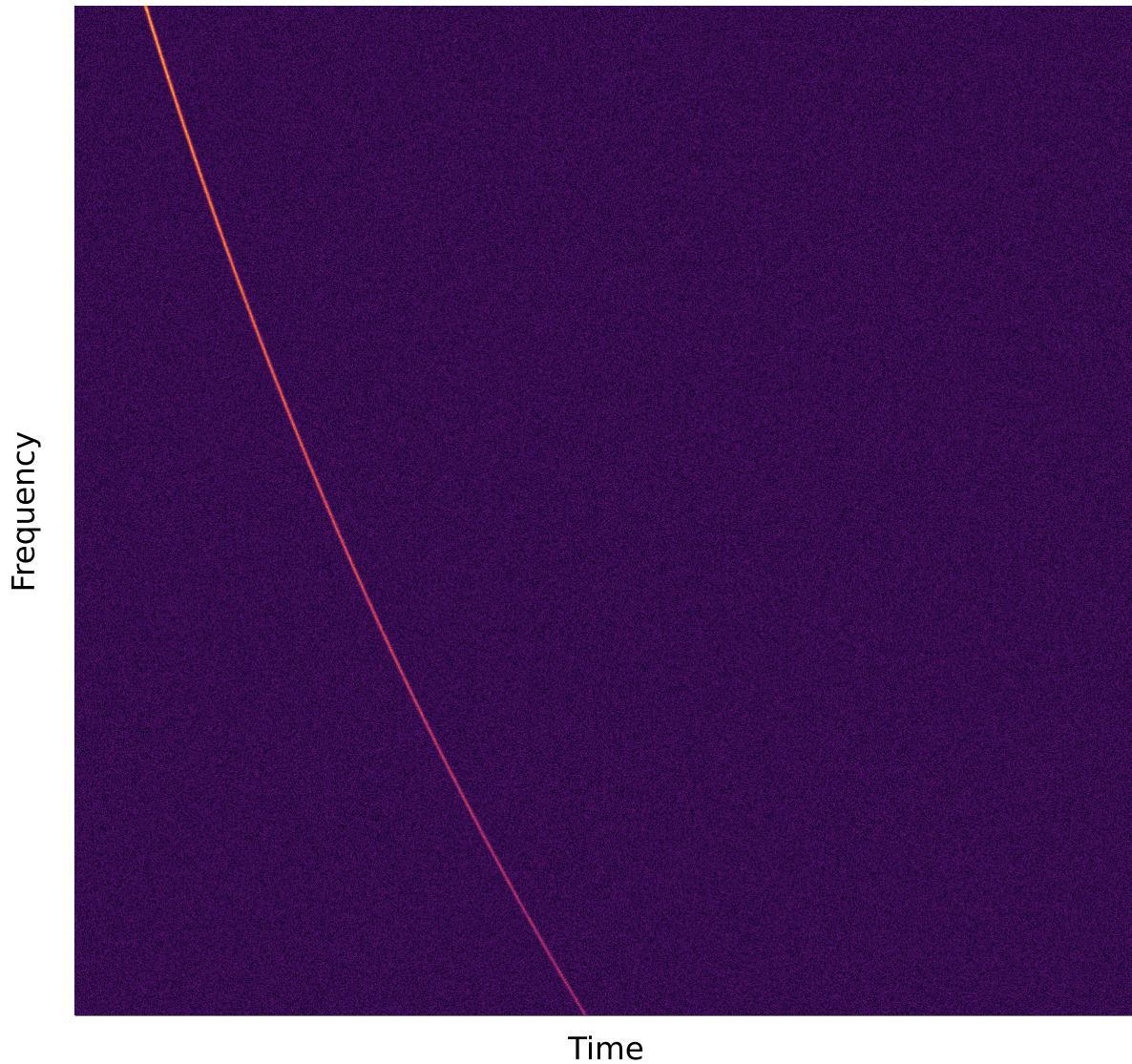
Benchmarking these results allows us to focus our reimplementation efforts into the most promising algorithms for use in doppler search pipelines. This work is necessary because **anecdotal evidence has suggested that existing dedoppler implementations are not performant enough to run on our future real-time commensal pipeline setups**. Additional work is currently being done to benchmark existing dedoppler implementations.

Test Input Data

This is the test dispersed pulse used for evaluation and benchmarking. This is the only input data used for this stage of comparison since we want to retool these algorithms to dedoppler linear-drifting signals before making more rigorous comparisons.

The input data size is 3000 time samples with 4096 frequency channels. For all of the implementations, 2076 unique DM trials were tested from 0 to 2000. Data transfer times between the CPU->GPU were not counted for GPU implementations. The intent is to compare the calculation time alone, but later considerations will take GPU overhead into account.

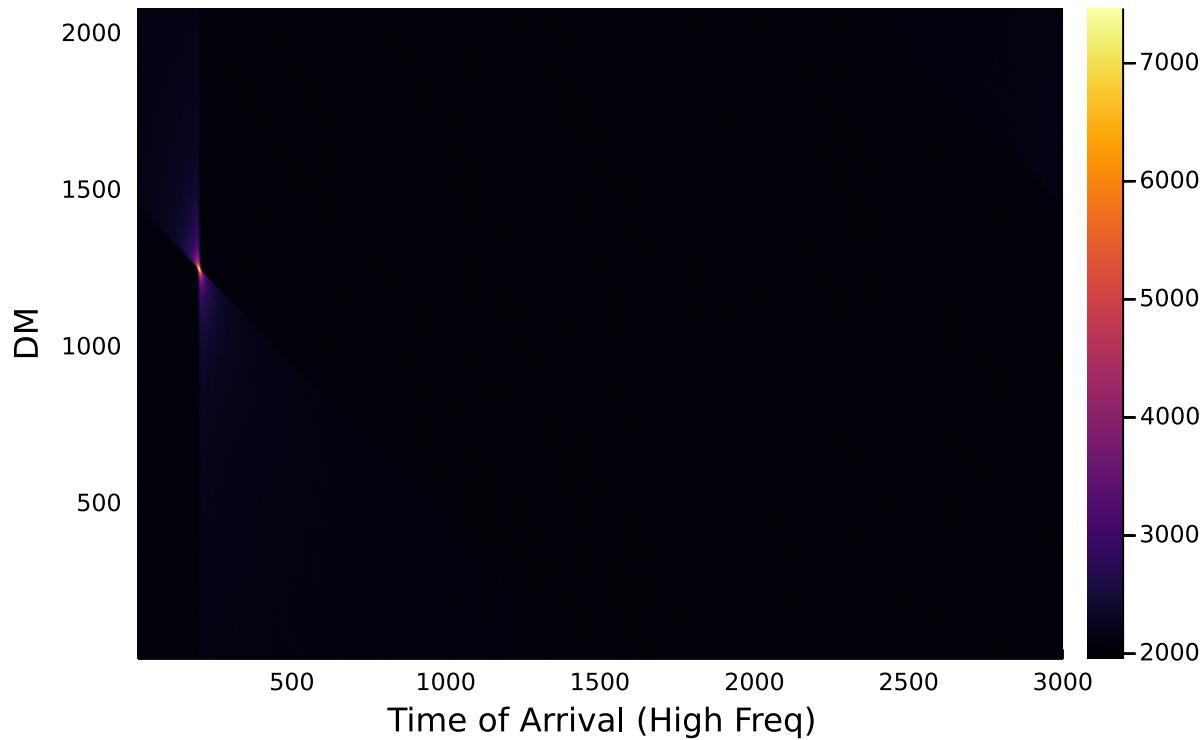
Dispersed Pulse



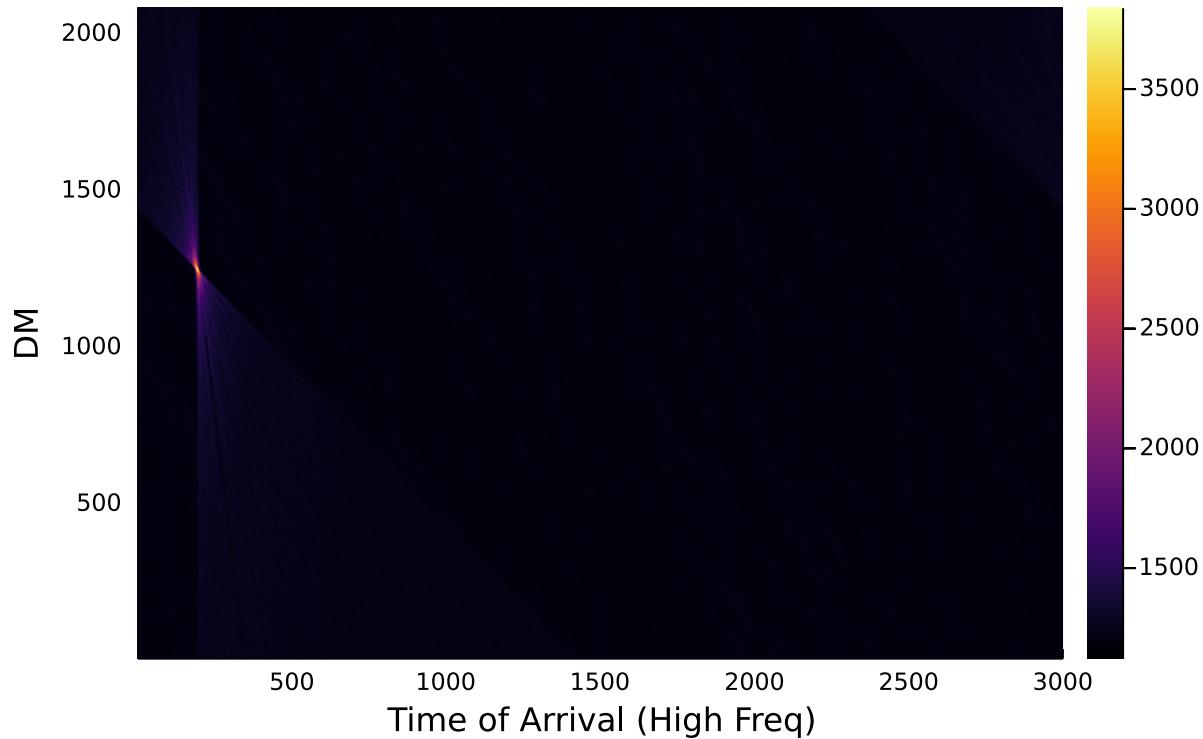
Output Verification

First, the above test data was fed through all the implementations and then their corresponding outputs were plotted below. A fair comparison requires that we first ensure that all the algorithms are doing roughly the same thing.

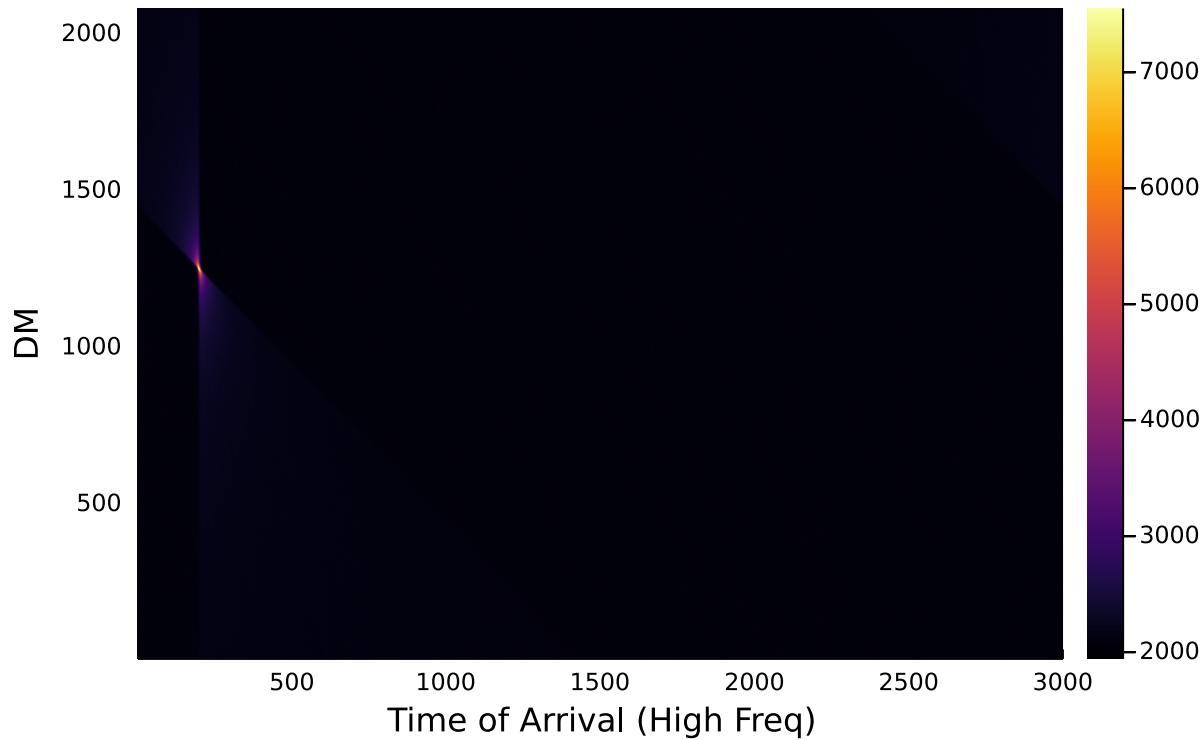
FDMT CPU Python Output



FDMT CPU Julia Output



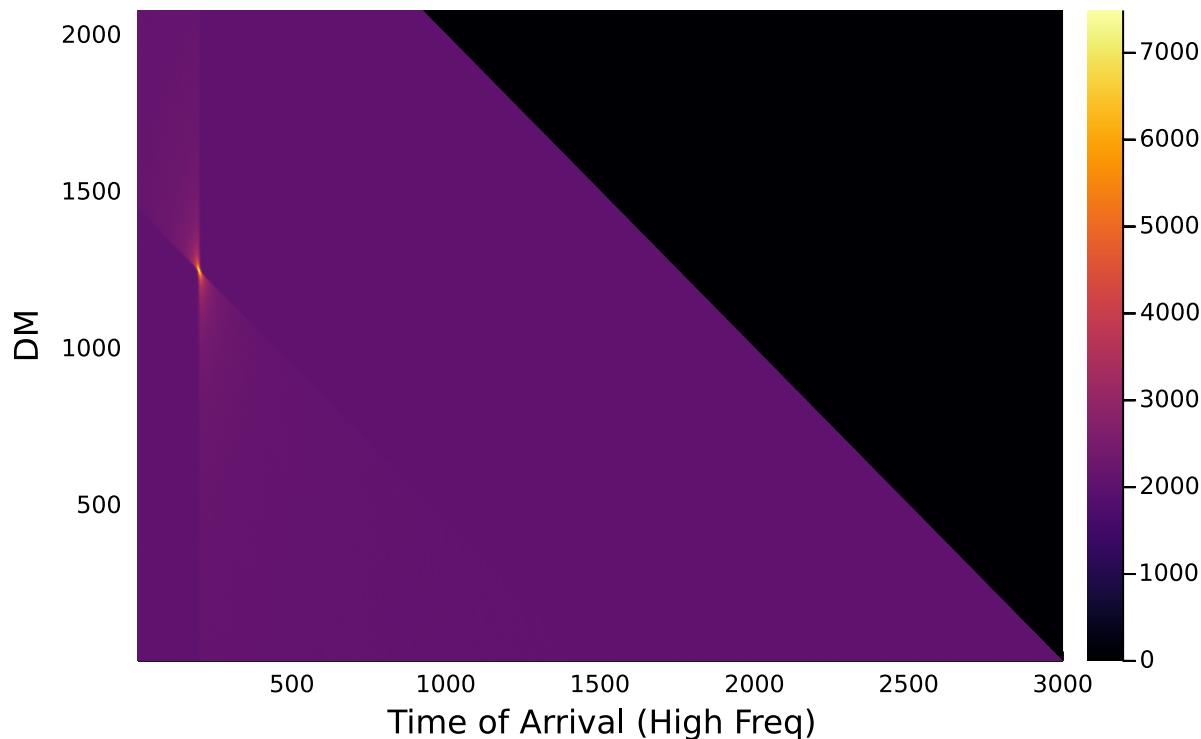
Dedisp.jl CPU Output



Dedisp.jl GPU Output



Bifrost FDMT GPU Output



As you can see, the outputs visually are very similar. These results, along with some numerical verification steps were enough to allow us to proceed with the benchmarks.

The only discrepancies are found in the FDMT CPU Julia and the FDMT GPU implementations. The CPU Julia version had minor discrepancies, but since the GPU versions were of primary interest in this evaluation, we didn't spend much time looking into this. The FDMT GPU implementation's output matches the others except for the upper right triangle which is filled with zeros.

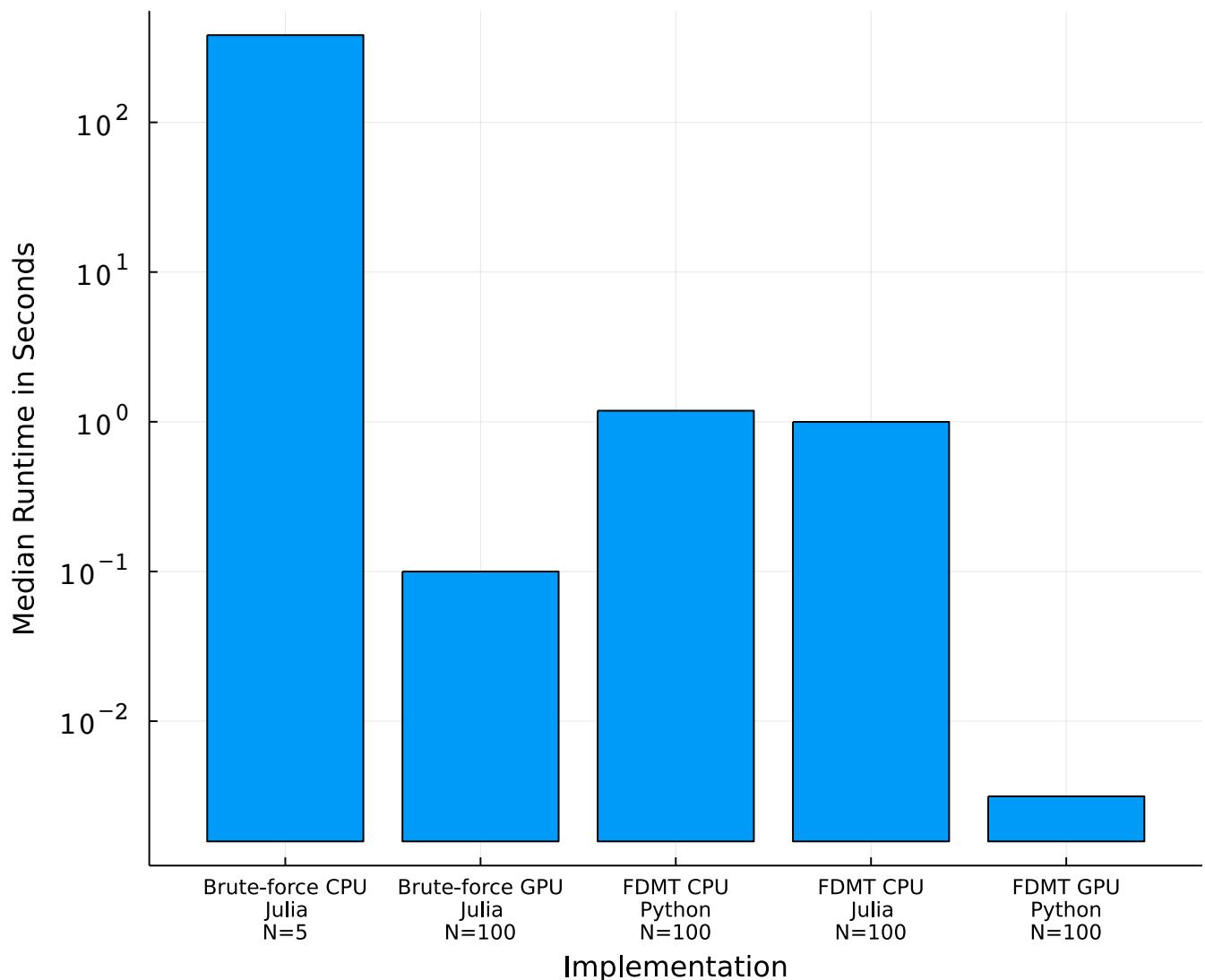
This region represents DM and time of arrival combinations that are not valid for the limited input given because the DM curve wraps past the extent of the current block. Values in this region are invalid and comparisons of results in this region are not meaningful.

Runtime Comparisons

Once the outputs were verified, each algorithm was run N times while recording the runtime for each trial.

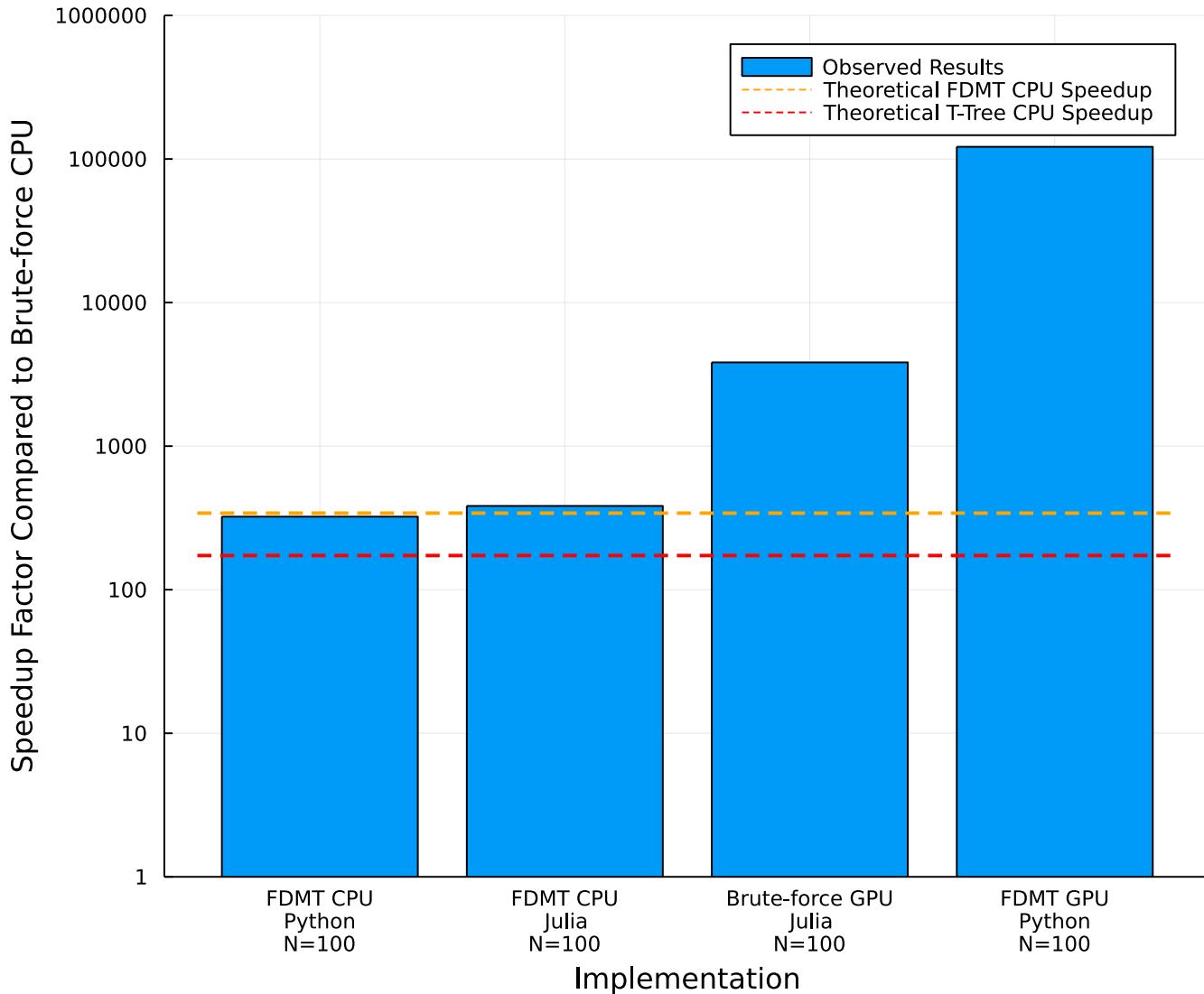
The following benchmark data were collected on Breakthrough Listen's blpc1 server using a Titan Xp GPU. If you have additional questions about the setup for these benchmarks, contact Max Hawkins. All benchmarks should be taken with some grains of salt, but the intent of this project was to evaluate the general nature and performance of existing dedoppler implementations. Relative order of magnitude comparisons are what we aimed for.

Dedispersion Runtime Comparison



GPU-acceleration and the FDMT algorithms gave drastic speedups. These results match the expected behavior based on algorithmic complexity analysis and the parallel nature of GPUs.

Dedispersion Runtime Comparison



Here, all runtimes were graphed relative to the slowest implementation (brute-force CPU). As you can see, the Python CPU FDMT implementation speedup almost exactly matches the expected theoretical algorithmic complexity speedup. The Julia implementation is higher than expected likely due to compiler tricks, vectorized instructions, or a highly inefficient brute-force CPU implementation.

Conclusion

For future dedispersion algorithm reimplementation for doppler search pipelines, the FDMT GPU algorithm should be the implementation of focus.

Future Work

In the future, we'd like to also evaluate the performance of the following implementations (primarily Fourier domain dedispersion) as the literature and exterior benchmarks show favorably even against the GPU FDMT implementation from this study:

GPU Brute-Force Dedispersion in C/C++ (Dedisp library)

Codebase: <https://github.com/kiranshila/dedisp>

GPU Fourier Domain Dedispersion in C/C++ (Dedisp library)

Codebase: <https://github.com/kiranshila/dedisp>

Run these benchmarks yourself

The repository containing these benchmark scripts and post-processing steps can be found here:
<https://github.com/max-Hawkins/bl-dedoppler-compare>

This document was created with Pluto.jl from src/final_comparison.jl. That file can be run reactively like a notebook (similar to Jupyter) but can also be run from the command line as a script.

July 1st, 2022 - Max Hawkins and Kevin Jordan with help from Dave MacMahon, Daniel Czech, and Kiran Shila (CalTech)