AstroAccelerate

GPU accelerated signal processing on the path to the Square Kilometre Array

Wes Armour, Karel Adamek,
Sofia Dimoudi, Jan Novotny, Nassim Ouannough, Cees Carels

Oxford e-Research Centre,
Department of Engineering Science
University of Oxford
20th March 2019

Part One

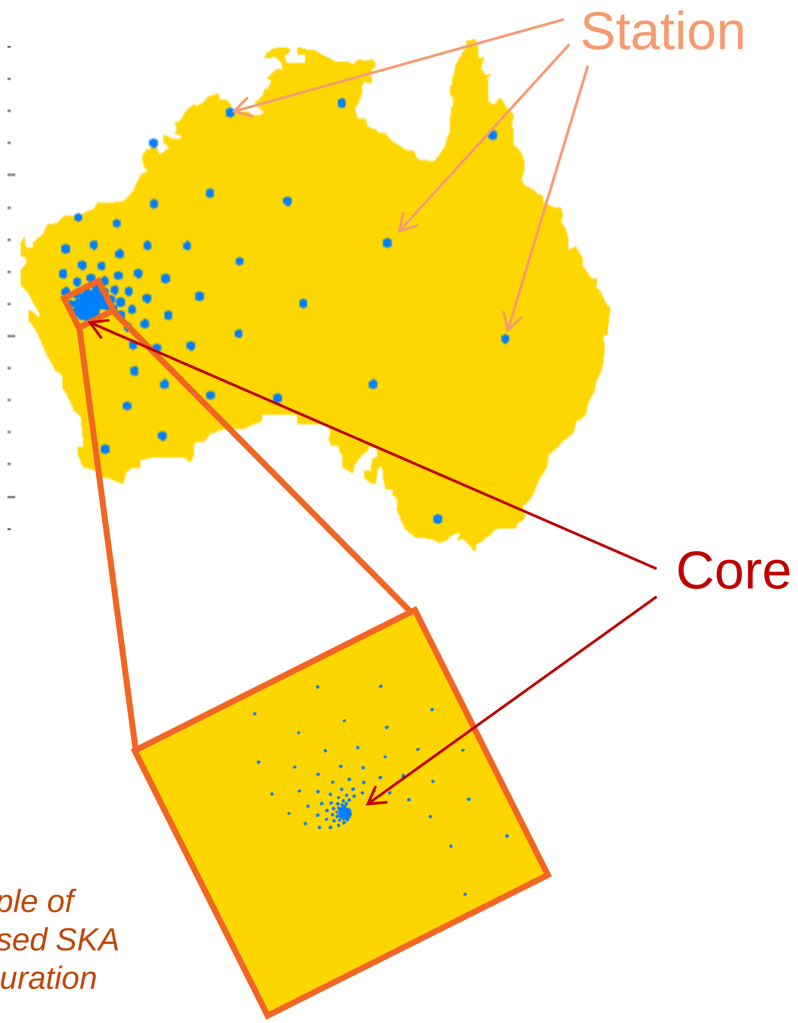
A brief introduction to

SQUARE KILOMETRE ARRAY

UNIVERSITY OF OXFORD

Oxford e-Research Centre

What is SKA?



Example of proposed SKA configuration

What does SKA stand for?

Square Kilometre Array, so called because it will have an effective collecting area of a square kilometre.

What is SKA?

SKA is a ground based radio telescope that will span continents.

Where will SKA be located?

SKA will be built in South Africa and Australia.

Graphic courtesy of Anne Trefethen

SKA science

SKA will study a wide range of science cases and aims to answer some of the fundamental questions mankind has about the universe we live in.

- How do galaxies evolve
  - What is dark energy?
- Tests of General Relativity
  - Was Einstein correct?
- Probing the cosmic dawn
  - How did stars form?
- The cradle of life
  - Are we alone in the Universe?

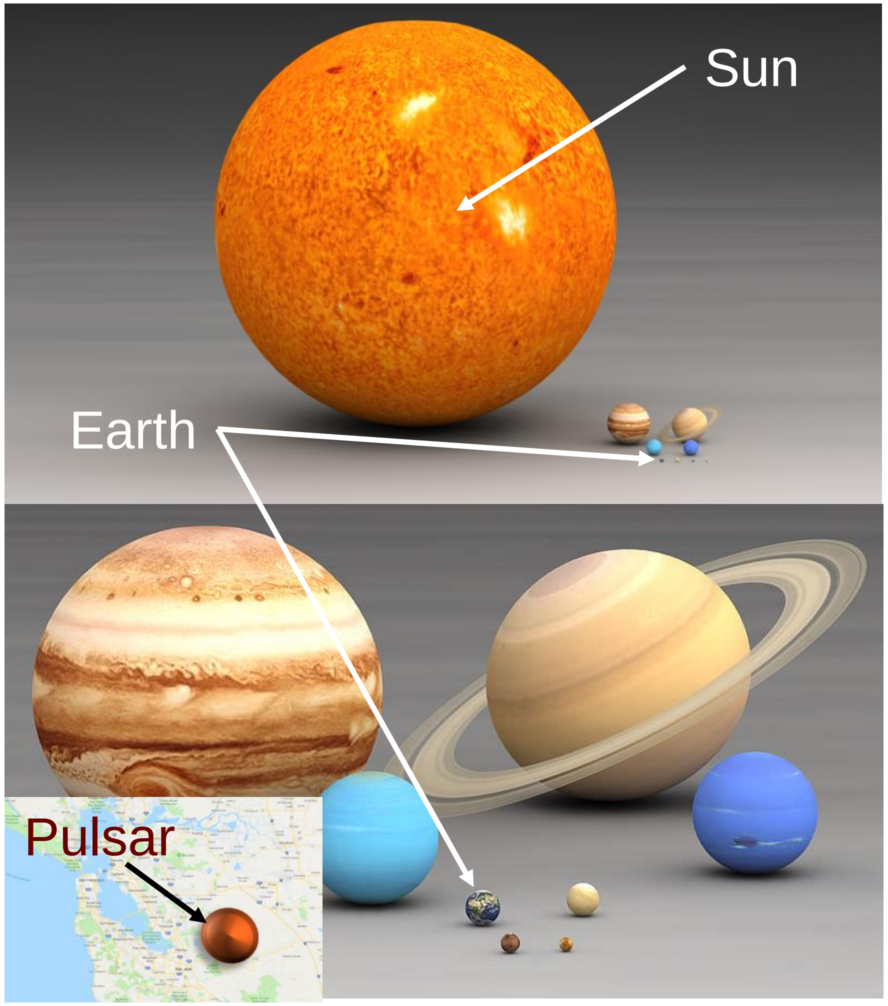
UNIVERSITY OF OXFORD

Oxford e-Research Centre

Part Two

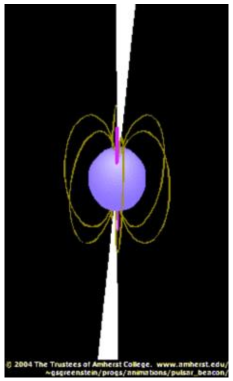
Time domain science

Pulsars – size and scale



Pulsars are magnetized, rotating neutron stars which emit synchrotron radiation from their poles (Crab Nebula). They are typically 1-3 Solar masses in size, have a diameter of 10-20 Kilometres and a pulse period ranging from milliseconds to seconds.

Their magnetic field is offset from the axis of rotation so we observe them as cosmic lighthouses.

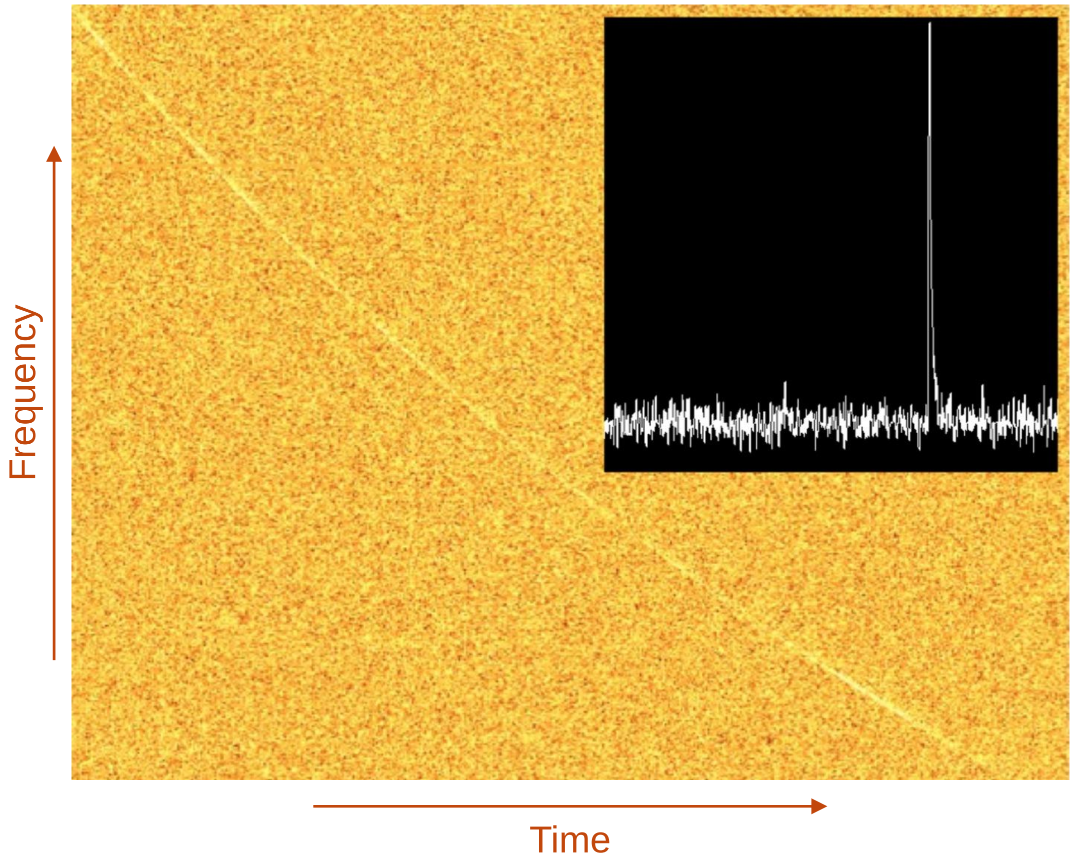

© 2004 The Trustees of Amherst College. www.amherst.edu/~gsgreenstein/progs/animations/pulsar_beacon/

Amherst College

Hester et. al.

https://commons.wikimedia.org/wiki/File:Planets_and_sun_size_comparison.jpg (Author: Lsmpascal)

SKA time domain science - Fast Radio Bursts



Fast Radio Bursts (FRBs), were first discovered in 2005 by Lorimer et al.

They are observed as extremely bright single pulses that are extremely dispersed (meaning that they are likely to be far away, maybe extra galactic).

So far around 15 have been observed in survey data. They are of unknown origin, but likely to represent some of the most extreme physics in our Universe.

Hence they are extremely interesting objects to study.

Credit: FRB110220 Dan Thornton (Manchester)

Part Three

Computing challenges

SKA time domain - data rates

The SKA will produce vast amounts of data. In the case of time-domain science we expect the telescope to be able to place ~**2000 observing beams** on the sky at any one time (there are trivially parallel to compute).

The telescope will take **20,000 samples per second for each of those beams** and then it will measure power in **4096 frequency channels for each time sample**. Each of those individual samples will comprise of 4x8 bits, although we are only really **interested in one of the 8 bits of information**.

Doing the math tells us that we will need to **process 160GB/s of relevant data**. This is approximately equal to analysing 50 hours of HD television data per second.

The most costly computational operations in data processing pipeline are

DDTR ~ $O(n_{dms} * n_{beams} * n_{samps} * n_{chans})$

FDAS ~ $O(n_{dms} * n_{beams} * n_{samps} * n_{acc} * \log(n_{samps}) * 1/t_{obs})$

Requiring ~2 PetaFLOP of Compute!

UNIVERSITY OF OXFORD

Oxford e-Research Centre

SKA time domain - signal processing

The time domain team is an international team led by Oxford and Manchester.

It aims to deliver an end-to-end signal processing pipeline for time domain science performed by SKA (see right).

Our work at OeRC has focussed on vertical prototyping activities. We are interested in using many-core technologies, such as GPUs to perform the processing steps within the signal processing pipeline with the aim of achieving real-time processing for the SKA.

Time Domain Team

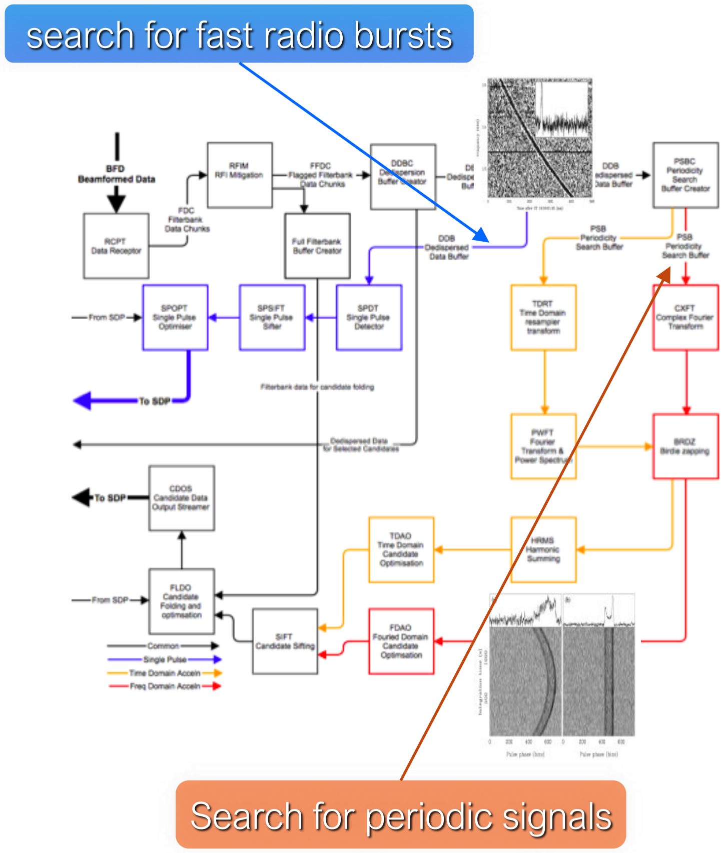

search for fast radio bursts
BFD Beamformed Data
RCPT Data Receptor
FDC Filterbank Data Chunks
RFIM RFI Mitigation
FFDC Flagged Filterbank Data Chunks
DDBC Dedispersion Buffer Creator
Full Filterbank Buffer Creator
DDB Dedispersed Data Buffer
PSBC Periodicity Search Buffer Creator
PSB Periodicity Search Buffer
SPDT Single Pulse Detector
SPSIFT Single Pulse Sifter
SPOPT Single Pulse Optimiser
From SDP
To SDP
Filterbank data for candidate folding
Dedispersed Data for Selected Candidates
TDRT Time Domain resampler transform
CXFT Complex Fourier Transform
PWFT Fourier Transform & Power Spectrum
BRDZ Birdie zapping
HRMS Harmonic Summing
TDAO Time Domain Candidate Optimisation
FDAO Fouried Domain Candidate Optimsation
SIFT Candidate Sifting
FLDO Candidate Folding and optimsation
CDOS Candidate Data Output Streamer
Common
Single Pulse
Time Domain Acceln
Freq Domain Acceln
Search for periodic signals

Image courtesy of Aris Karastergiou

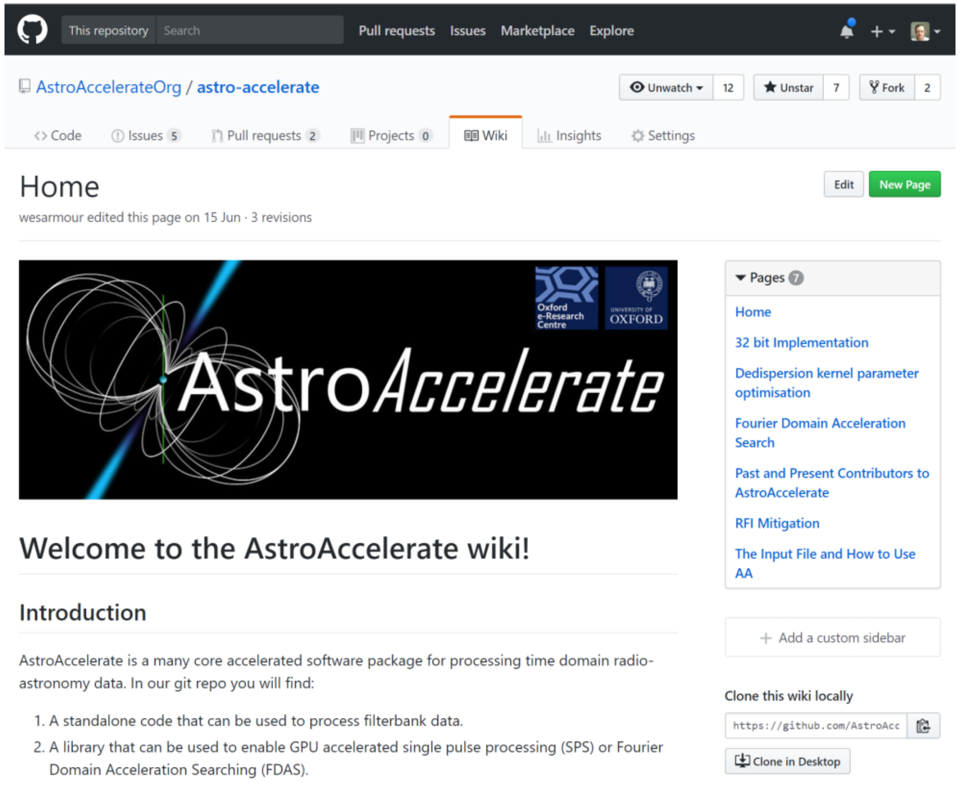
Part Four

GPU accelerated signal processing library for time-domain radio astronomy.

AstroAccelerate

AstroAccelerate is a GPU enabled software package that focuses on achieving real-time processing of time-domain radio-astronomy data. It uses the CUDA programming language for NVIDIA GPUs.

The massive computational power of modern day GPUs allows the code to perform algorithms such as de-dispersion, single pulse searching and Fourier Domain Acceleration Searching in real-time on very large data-sets which are comparable to those which will be produced by next generation radio-telescopes such as the SKA.



Home

wesarmour edited this page on 15 Jun · 3 revisions

Welcome to the AstroAccelerate wiki!

Introduction

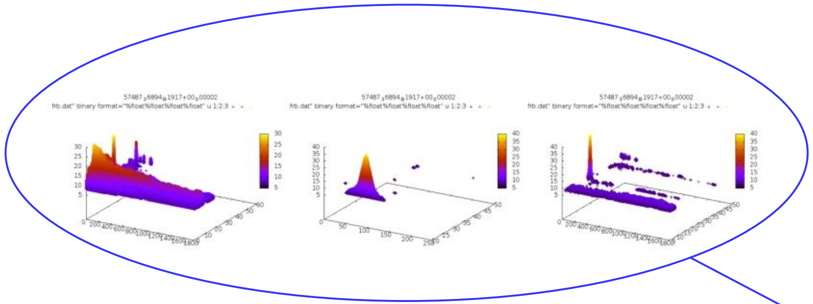
AstroAccelerate is a many core accelerated software package for processing time domain radio-astronomy data. In our git repo you will find:

1. A standalone code that can be used to process filterbank data.
2. A library that can be used to enable GPU accelerated single pulse processing (SPS) or Fourier Domain Acceleration Searching (FDAS).

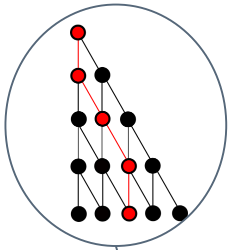
https://github.com/AstroAccelerateOrg/astro-accelerate

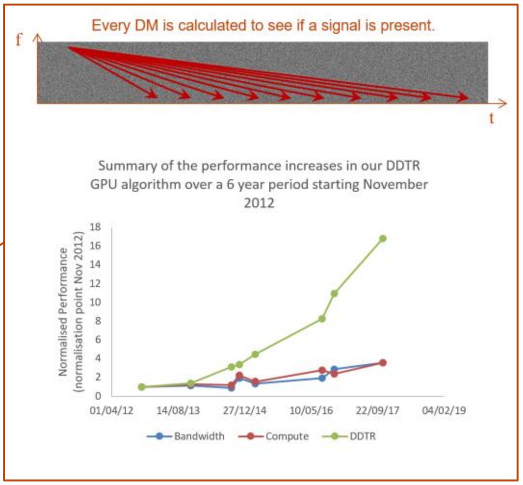
AstroAccelerate - Signal Processing

Radio Frequency Interference Mitigation



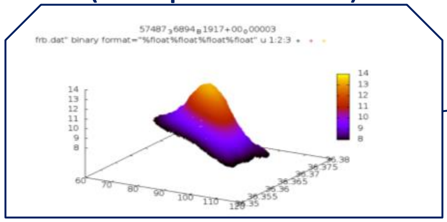
Harmonic Sum
(Deep dive two)

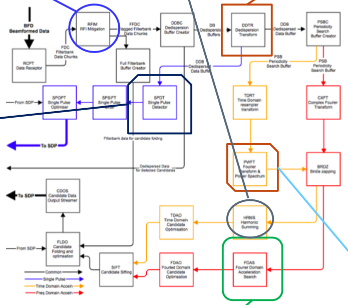



Every DM is calculated to see if a signal is present.
f
t
Summary of the performance increases in our DDTR GPU algorithm over a 6 year period starting November 2012
Normalised Performance (normalisation point Nov 2012)
01/04/12
14/08/13
27/12/14
10/05/16
22/09/17
04/02/19
Bandwidth
Compute
DDTR

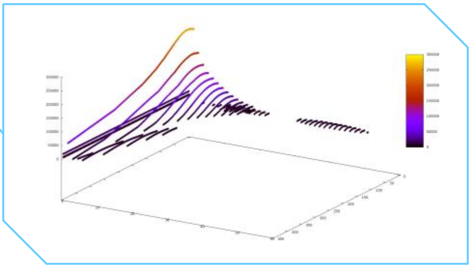
De-dispersion

Single Pulse Search
(Deep dive one)

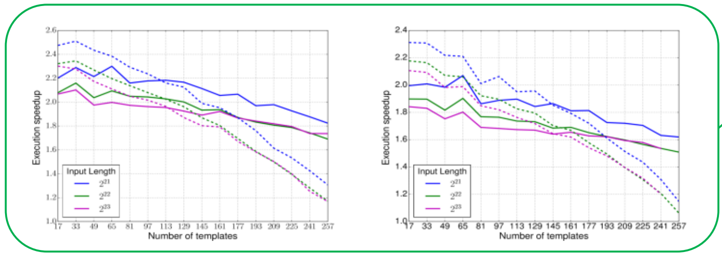




Periodicity Search

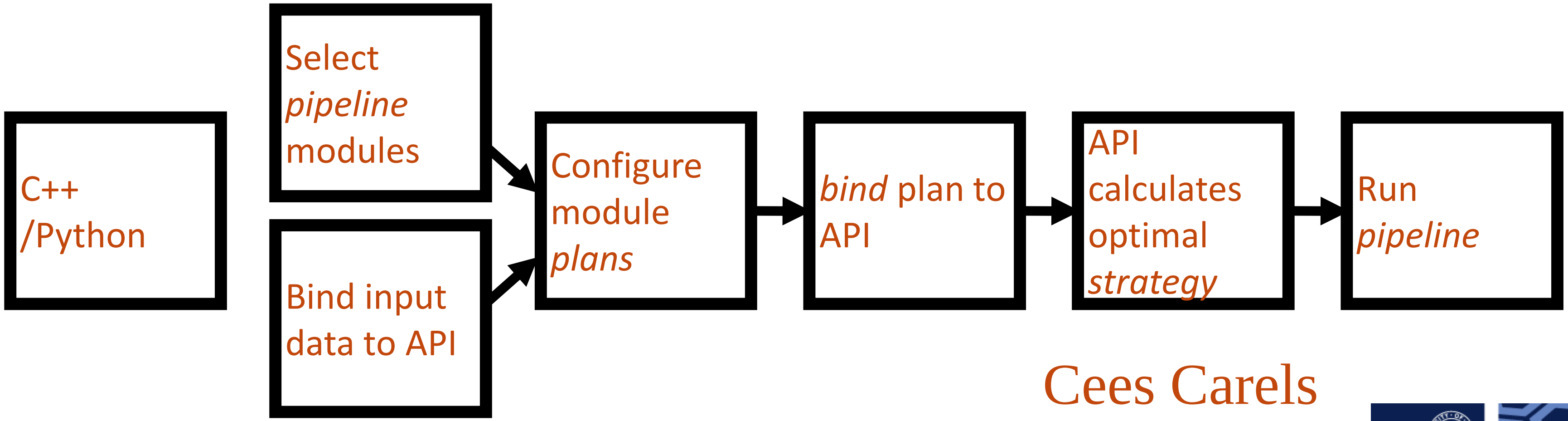


Fourier Domain Acceleration search



AstroAccelerate - API

- API follows a simple pattern: *configure, bind, run*.
- Select which *pipeline* modules to run, configure module *plan*, then *bind* plan to the API.
- API calculates the *strategy* with the optimal configuration for the *plan*.
- When all *strategy* objects are ready, the user selected modules are run within a *pipeline*.



Cees Carels

AstroAccelerate - Code Features

- Usable as a library (.so) and/or standalone executable.
- Examples with instructions on how to compile and link.
- Regular releases (semantic versioning).
- CMake build system.
- Full doxygen documentation and readme.
- Automated CI, unit tests.

Part Five

Deep dive into recent work

Single Pulse Detection

Karel Adámek, Wes Armour

Single Pulse Search

Aim is to detect pulses of different shapes and widths at unknown position within the signal and do it quickly.

Single pulse search (SPS) could be done through matched filters these are very sensitive but has problem with “quickly”.

Using a Boxcar filter for the single pulse search (SPS):

- Allows us to reuse data
- Independent of pulse shape
- We can trade sensitivity for performance
- Less sensitive by design

Single Pulse Search: How to detect pulses with boxcars

Signal's strength is measured as signal-to-noise ratio (SNR)

$$SNR = \frac{x-\mu}{\sigma},$$

Where x is the sample value, μ is the mean and σ is the standard deviation.

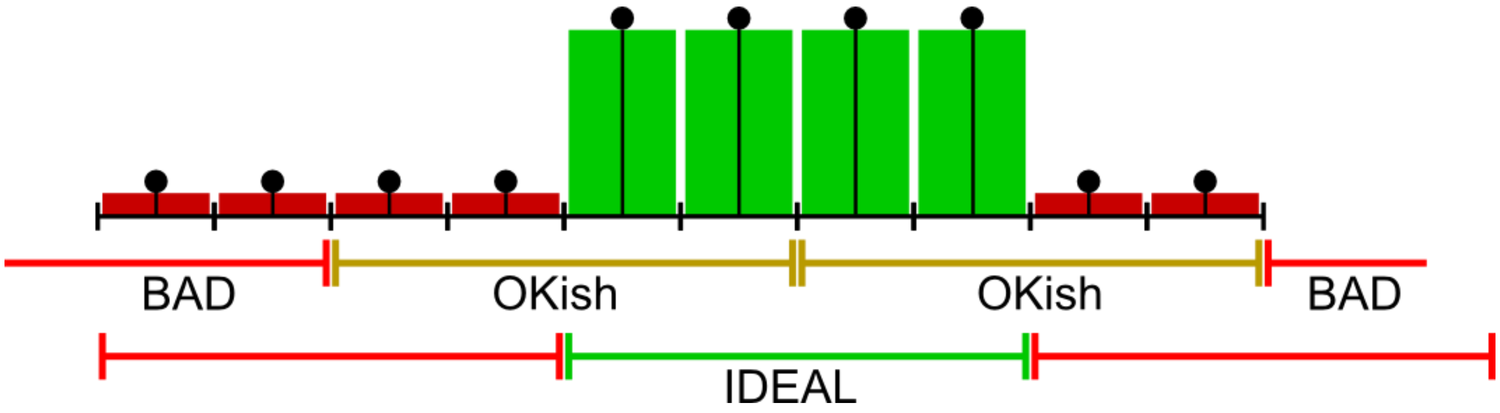
SNR is

- Increased by adding signal
- Decreased by adding noise

Position of the boxcar is important

We quantify coverage of the pulses by the distance between boxcar filters *L.*

- Pulse may end up between boxcars
- By decreasing L we cover pulses better



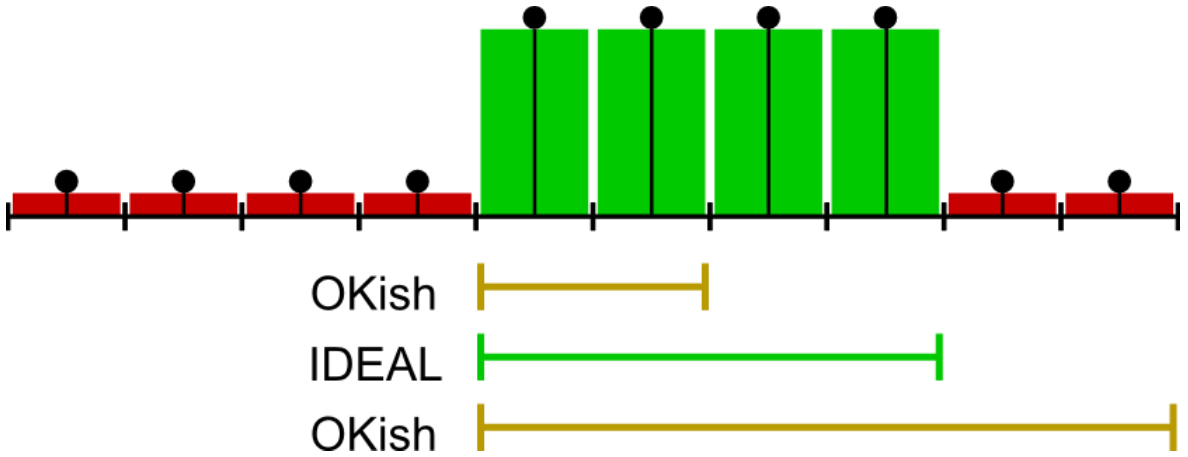
Single Pulse Search: How to detect pulses with boxcars

SNR is
- Increased by adding signal
- Decreased by adding noise

Boxcar which is:
- too short does not cover pulse fully
- too long does add unnecessary noise

Width of the boxcar filter is also important

We need different boxcar widths W to better detect different pulse widths.



Single Pulse Search: What do we need to do?

Summary:

- Position of the boxcar relative to the pulse is important. This is expressed by the distance between boxcars L.
- Boxcar width W is important for detection of pulses with different widths.

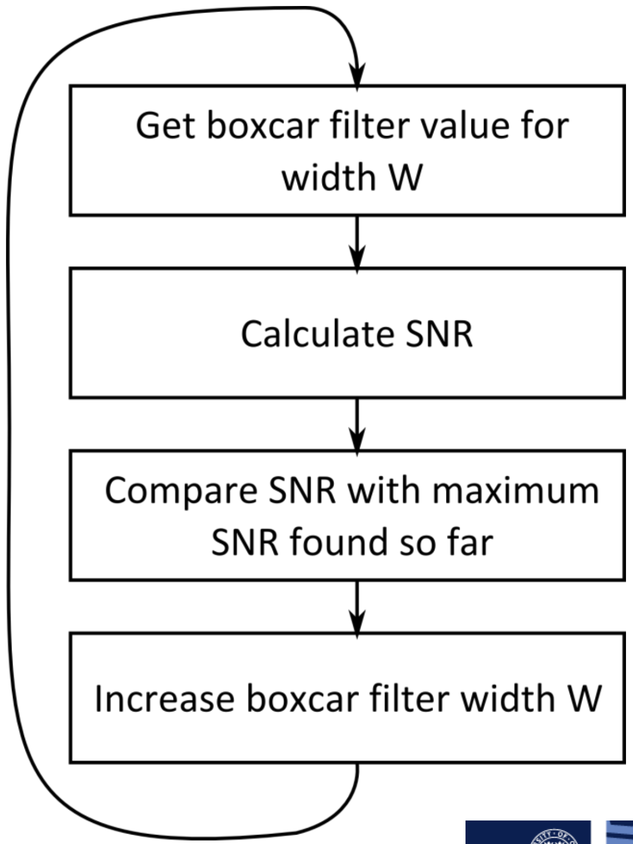
Output:

Highest SNR detected at given sample.

- We do not need to keep values of all boxcar filters just highest SNR!

For ideal detection we need to do:

at every point


Get boxcar filter value for width W
Calculate SNR
Compare SNR with maximum SNR found so far
Increase boxcar filter width W

Single Pulse Search: Two algorithms

How to adjust sensitivity
… and increase performance:

- By decreasing/increasing distance between boxcars L
- By performing more/less boxcars of different widths W
- After some point it is pointless to decrease L without more widths W

The algorithm must be able to
- perform very long boxcar filters; for SKA this is 8000+ samples
- Adjustable sensitivity

BoxDIT
- Starts from ideal Boxcar filter
- Top-down – starts with good sensitivity but poor performance
- Easily adjustable
- Can be very sensitive
- Not as fast

IGrid
- Start from decimation in time (DIT)
- Bottom-up – starts with good performance but poor sensitivity
- Less flexible
- Faster

Single Pulse Search: BoxDIT

BoxDIT has two steps:

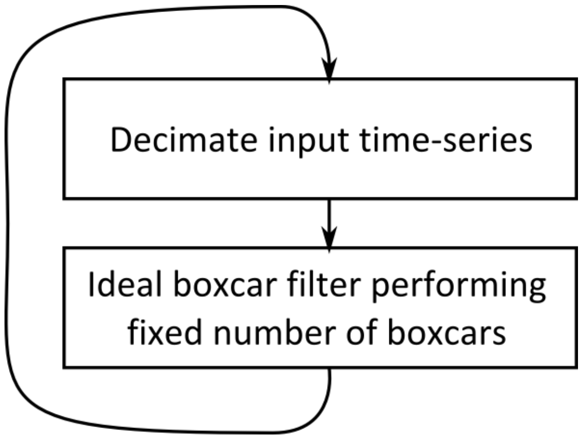
- Decimation in time - is used to control sensitivity
- Ideal boxcar filter (Scan) – is calculating boxcar filters.

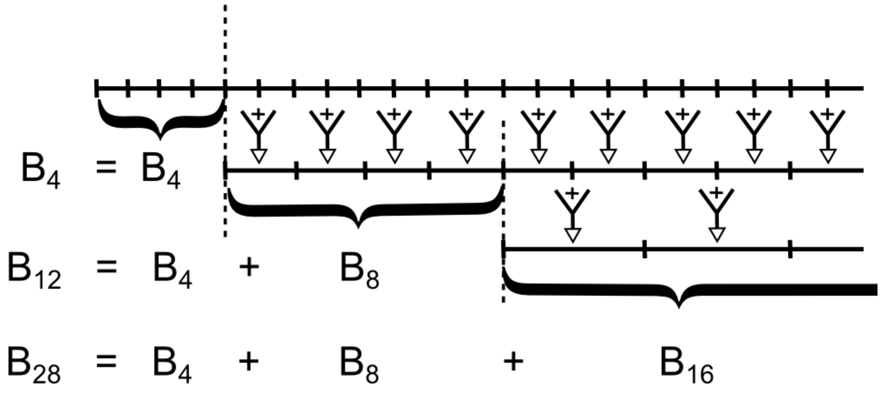
BoxDIT is reusing previously (time) decimated data to build longer boxcar widths.

In GPU implementation both steps are performed at once and kernel calculates boxcar filters as well as decimation for next iteration.

BOTTOM: Using combinations of data at different decimation levels allows us to construct longer width boxcars.

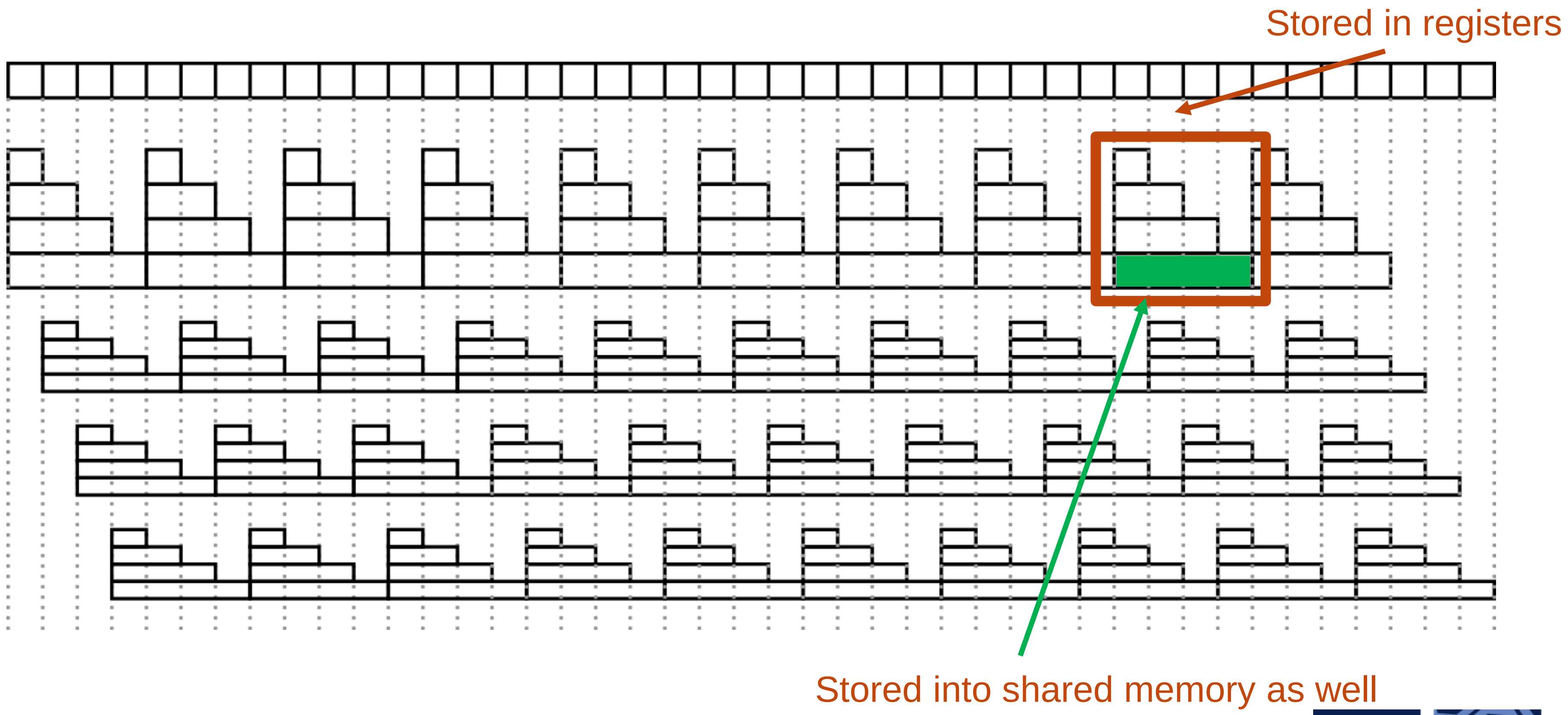
Diagram of the BoxDIT algorithm.


Decimate input time-series
Ideal boxcar filter performing fixed number of boxcars



Single Pulse Search: BoxDIT Scan at every point

Algorithm for scan at every point (applying set of boxcar filters) first calculate small scan at every point (here 4). The value of the longest boxcar (here 4) is stored into shared memory.



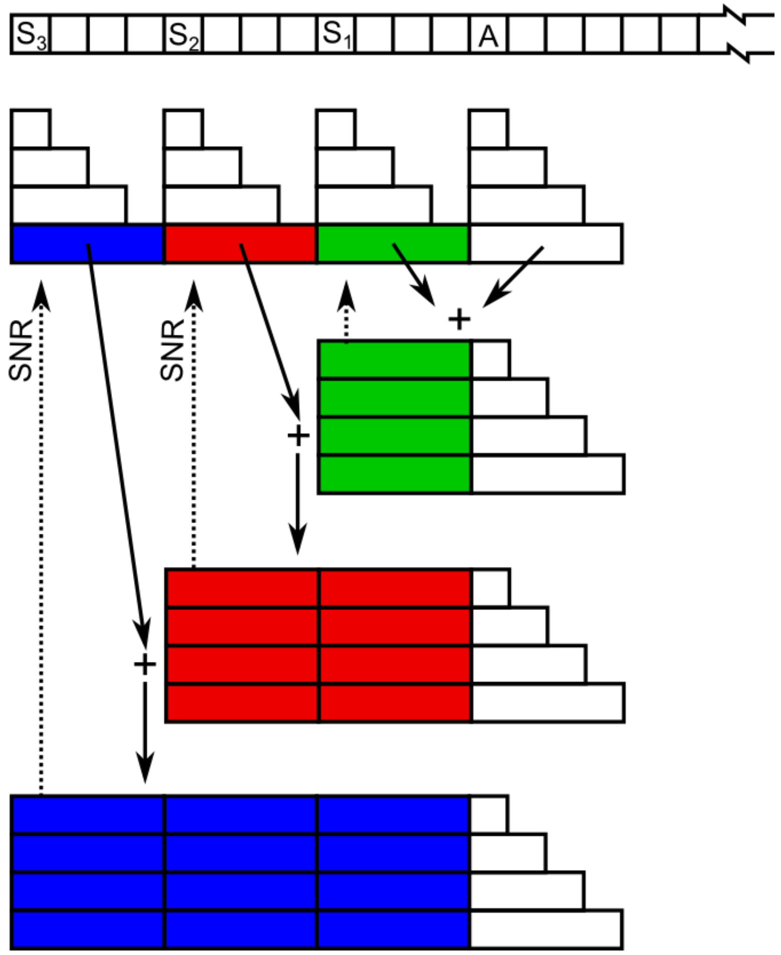
Single Pulse Search: BoxDIT scanpart

Showing algorithm steps only for every 4th thread. Other threads doing the same thing for other points.

Each thread keeps values of boxcar filters in registers. These are increased with every step of the algorithm.

In each step i, an active thread A, calculates a source thread id as $S_i = A - i*4$. The value of the longest boxcar calculated at beginning is loaded from source thread (from shared memory) and used to calculate longer boxcars in the active thread. These are kept by the active thread.

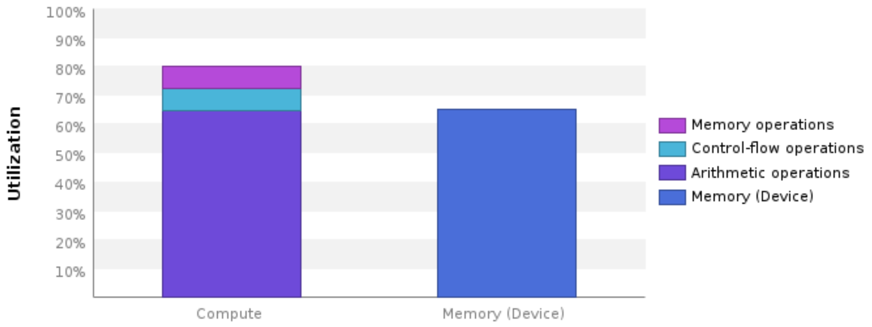
The highest SNR from the newly calculated boxcars is then compared with the SNR of the source thread and stored at it's position in shared memory if higher.



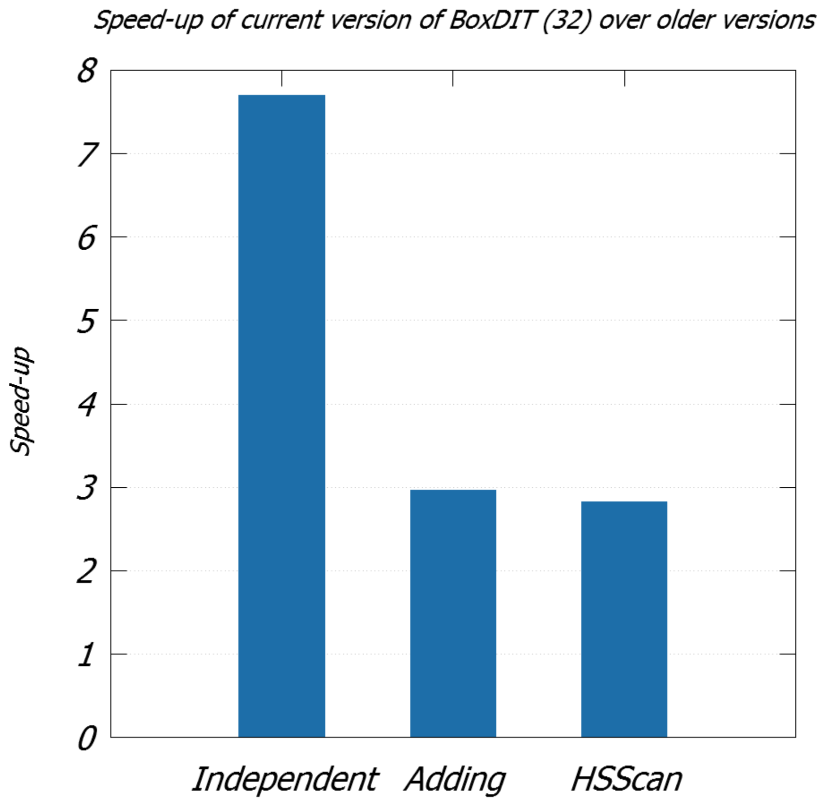
Single Pulse Search: BoxDIT performance

When calculating 32 boxcar per iteration (1% signal loss, idealised case) code is limited by compute with 63% device memory bandwidth utilisation. It is 83x faster then real-time.

When calculating 16 boxcars per iteration (2% signal loss, idealised) code is limited by device memory bandwidth (86%). It is 170x faster then real time.



Other versions of BoxDIT algorithm



Single Pulse Search: Two algorithms

BoxDIT

- Starts from ideal Boxcar filter
- Top-down – starts with good sensitivity but poor performance
- Easily adjustable
- Not as fast

IGrid

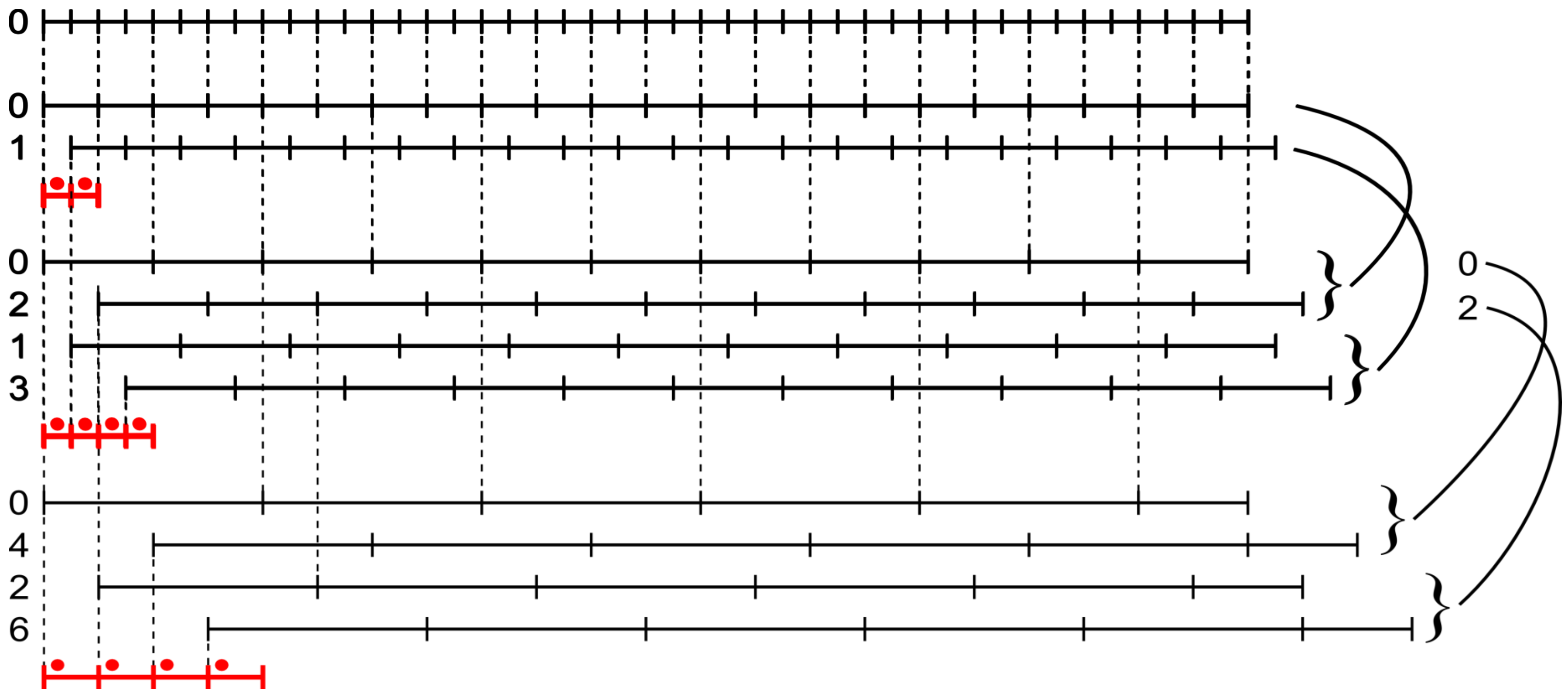
- Start from decimation in time (DIT)
- Bottom-up – starts with good performance but poor sensitivity
- Less flexible
- Faster

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Single Pulse Search: IGrid

The IGrid algorithm is based on combination of different decimations in time, which are shifted in time samples.

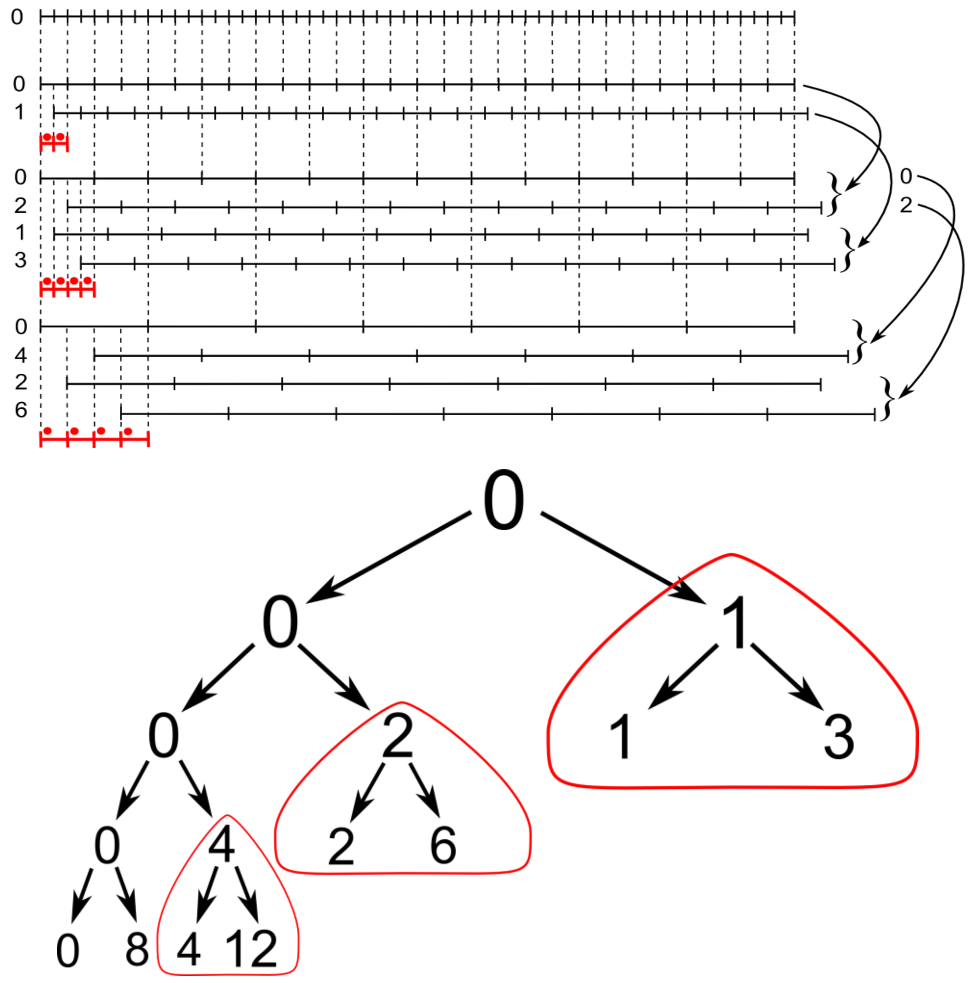


Single Pulse Search: IGRID

This algorithm could be interpreted also as a binary tree where leaves indicate a shift in number of time samples.

The binary tree view suggest what could be calculated locally (red area). Therefore the only thing shared through iterations is the time-series with zero shift.

It also offers a way how to calculate different boxcar widths, that is by going to the root of the tree.



UNIVERSITY OF OXFORD
Oxford e-Research Centre

Single Pulse Search: IGRID

To calculate individual IGRID iterations is inefficient and very demanding on device memory bandwidth.

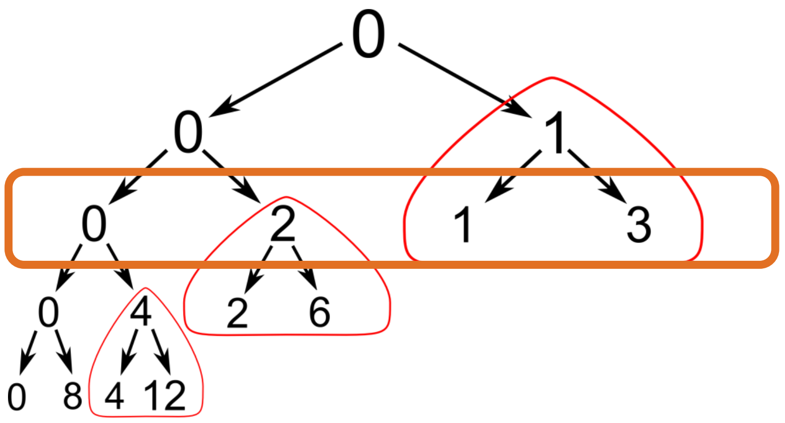
Thus we calculate multiple IGRID iterations per thread-block.

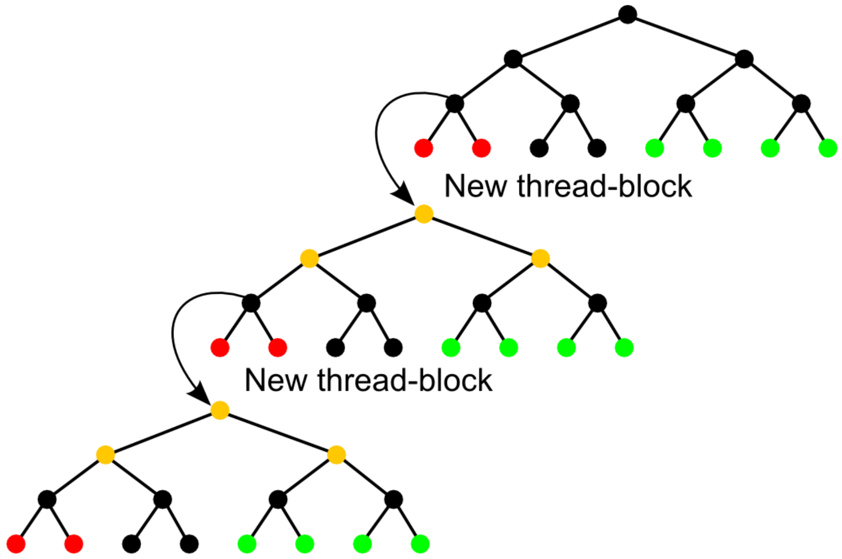
Points represent layers that

- ● must be calculated to get desired sensitivity
- ● must be calculated but it will be recalculated by the next block
- ● recalculated layer
- ● Is calculated but it is not required

TOP: to calculate individual IGRID iterations a lot of layers had to be shared

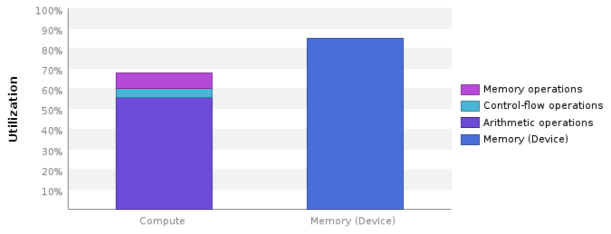
BOTTOM: Each thread-block calculates multiple iterations.



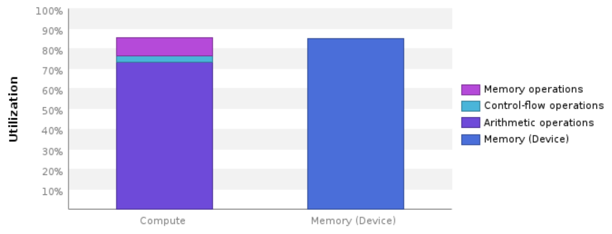


Single Pulse Search: BoxDIT performance

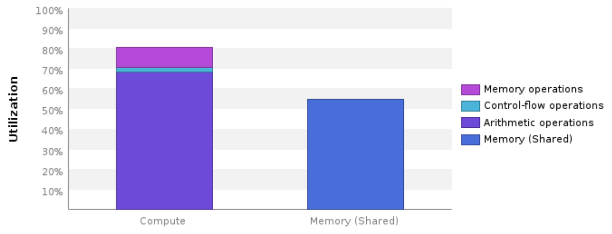
IGrid 1 – Signal loss ~6%

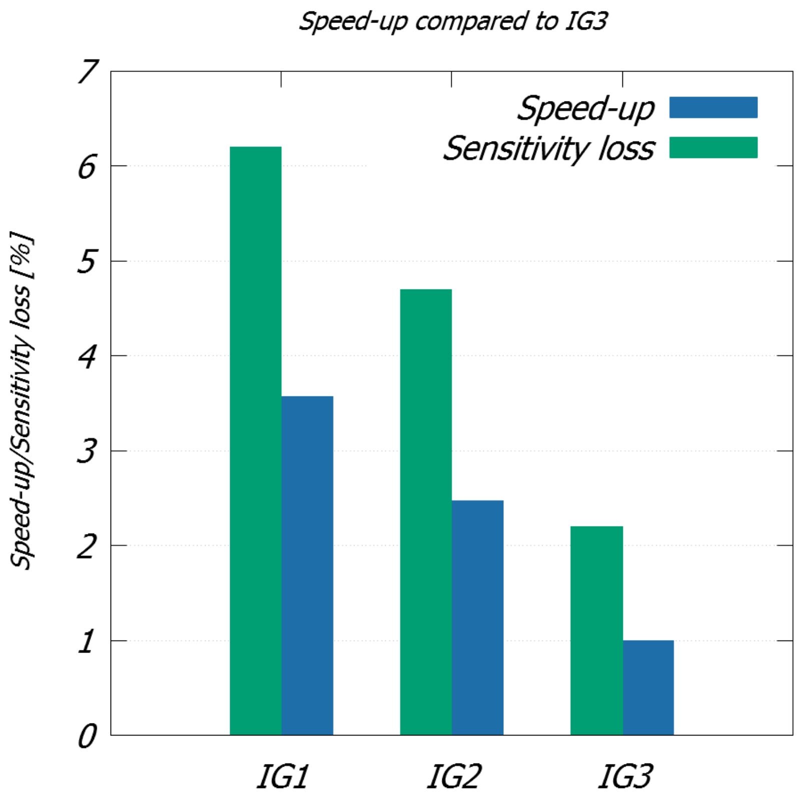


IGrid 2 – Signal loss ~4.5%

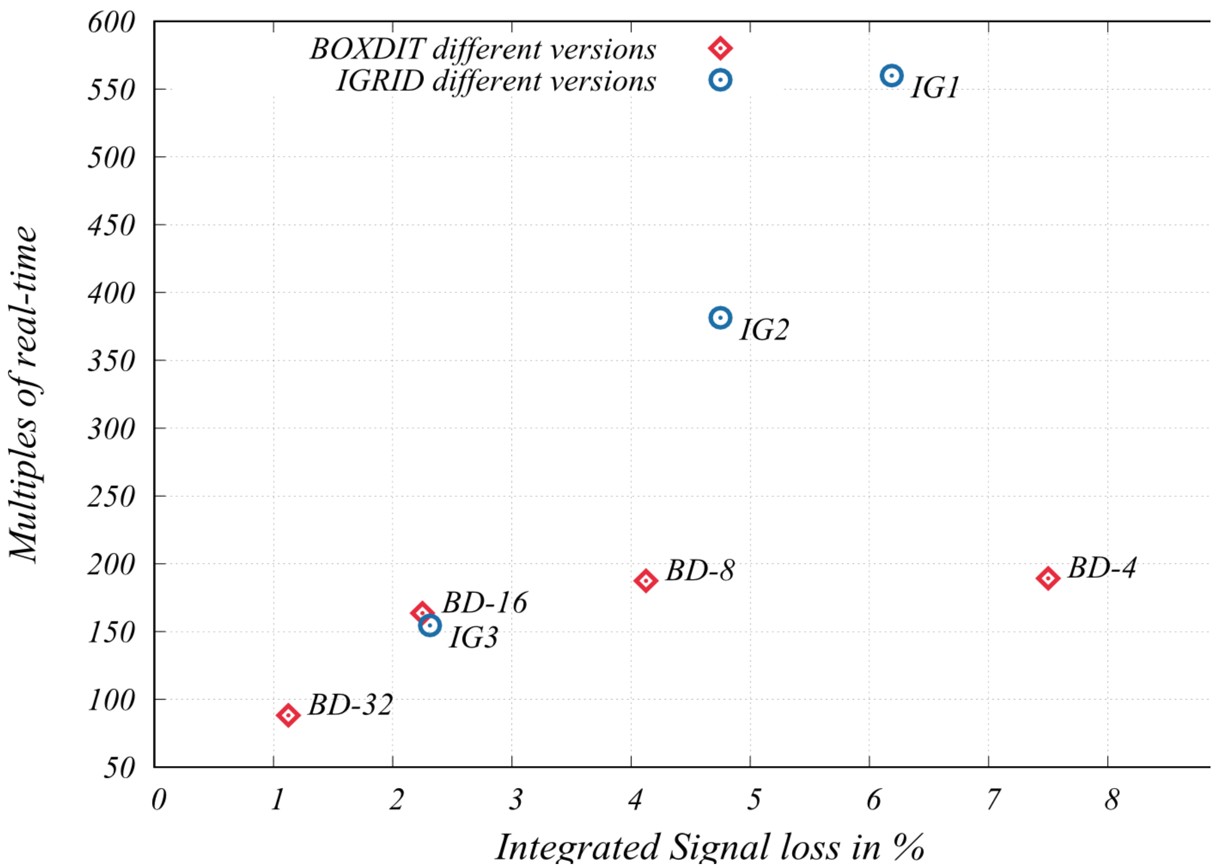


IGrid 3 – Signal loss ~2%





Single Pulse Search: Results



Number of DM trials/second for SKA-mid sized data is about ~6000.

This means BOXDIT is ~83x-200x faster then real time

IGRID is ~150x-580x faster then real time.

Left: Comparison of algorithms on average signal loss and performance (DM trials).

Conclusions

- Quantified source of sensitivity loss
- Adjustable sensitivity
- Two algorithms with different sensitivity/performance ratio
- BoxDIT algorithm is in AstroAccelerate and used for science output

Harmonic Sum

Karel Adámek, Jan Novotný, Wes Armour

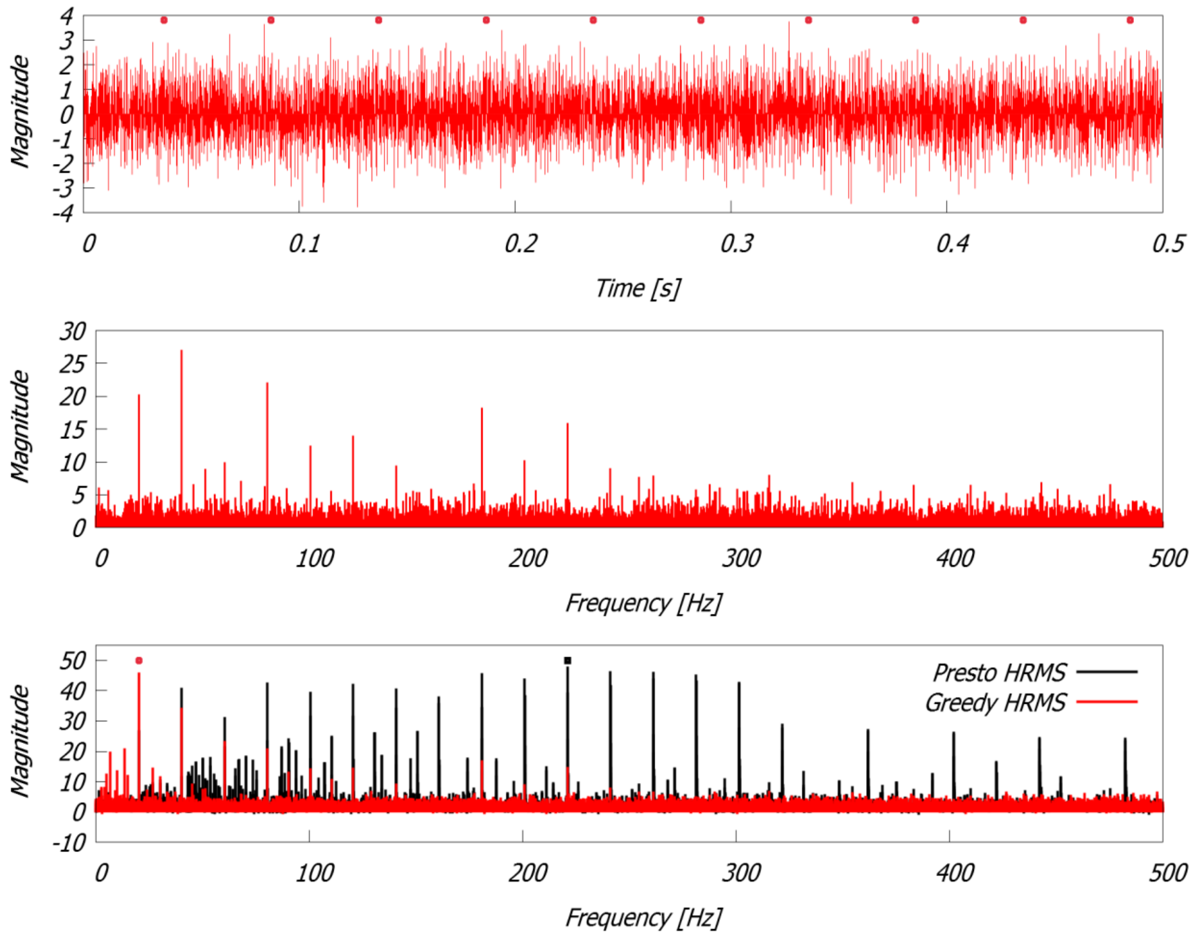
Harmonic sum

When searching for pulsars using Fourier domain methods the power of the pulsar is in frequency domain spread to multiple frequency bins. The incoherent harmonic sum algorithm is one way to correct this.

TOP: time-series containing pulsar (dots)

MIDDLE: frequency-domain harmonics visible

BOTTOM: result of harmonic sum for two diff. algorithms



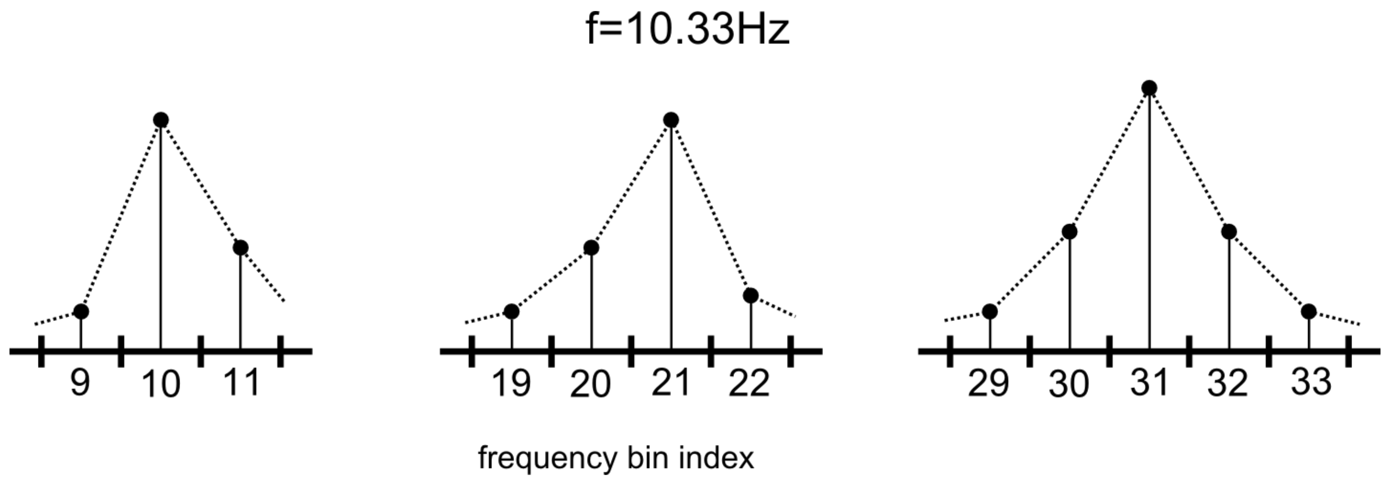
Harmonic sum

The goal of the harmonic sum is to sum pulsar's power that was spread into multiple harmonics which are integer multiples of the fundamental frequency f_0

$$\mathrm{h(n)}_H = \sum_{i=1}^{H} P(if_0)\,,$$

Where H is number of harmonics summed and P is the power spectrum.

But we do not know f_0 and we work with discrete indices



fundamental = 10.33Hz　first harmonic = 20.66Hz　second harmonic = 31Hz

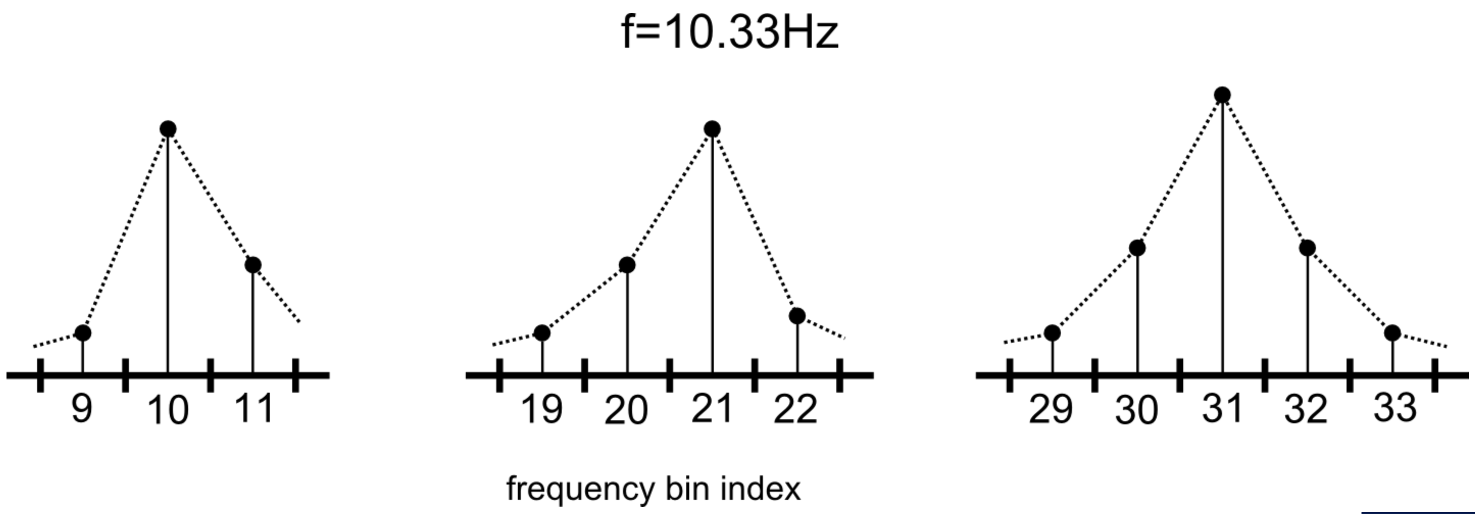
Harmonic sum

The defacto pulsar processing code, PRESTO, uses the following formula:

$$\mathrm{h(n)}_H = \frac{1}{\sqrt{H}} \sum_{i=1}^{H} P\left(\frac{ni}{H}\right)$$

H is number of harmonics summed.

You can approach it from different end. Start with the index position of the fundamental frequency *n* in frequency domain *X[n]* and add to it higher harmonics, that is *X[2n], …*



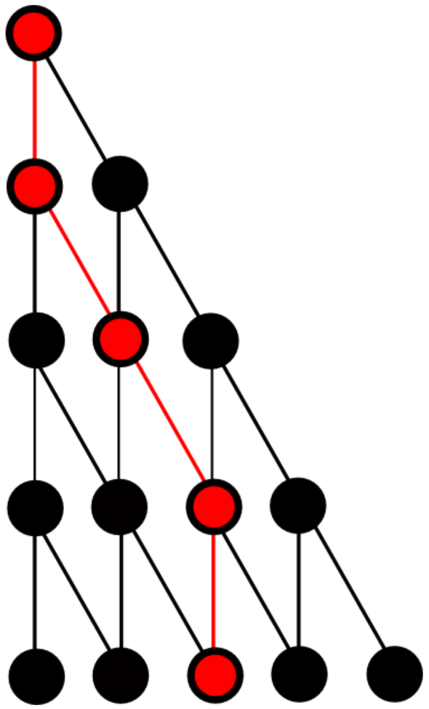
Harmonic sum – Problems

Harmonic sum – Problems

We aim to provide a selection of algorithm with different ratio of sensitivity/performance

- Unfavorable access pattern
    - h*(starting index)
    - Stride in memory access is *h*
    - Transpose of the data might help
- Difficult data reuse
    - Simple data reuse is limited
    - Cannot increase efficiency by increasing number of fundamental freq. bins
    - Complicated data reuse is resource and bookkeeping heavy
- Apply physical constrains
    - Decreases frequency range not index range which we need to explore.

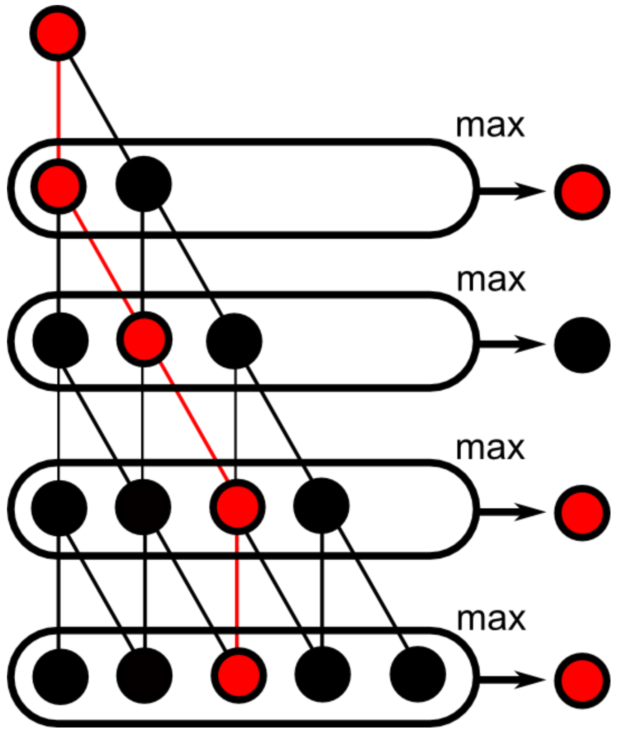
Harmonic sum – Tree view



There are few possibilities how to do harmonic sum

Create all possible sums (exhaustive search) best possible precision

Harmonic sum – Tree view

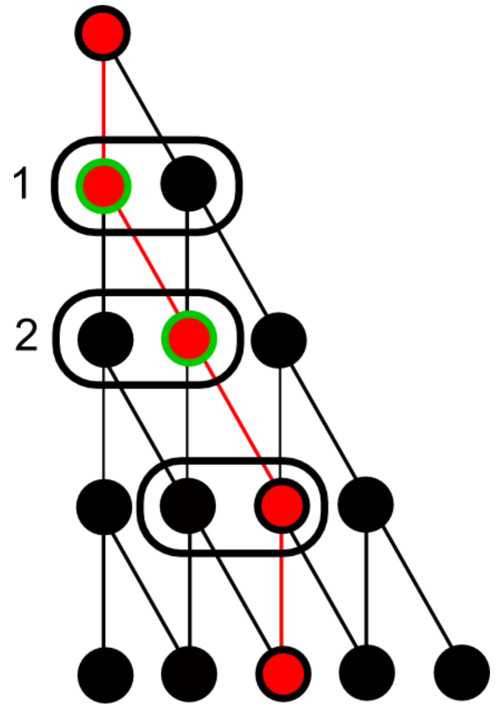


There are few possibilities how to do harmonic sum

Create all possible sums (exhaustive search) best possible precision

Construct the sum from maxima of each harmonic. That is $h(n)_H = \sum_{i=1}^{H} \max_{1 \le j \le i}\left(x(in+j)\right)$

Harmonic sum – Tree view



There are few possibilities how to do harmonic sum

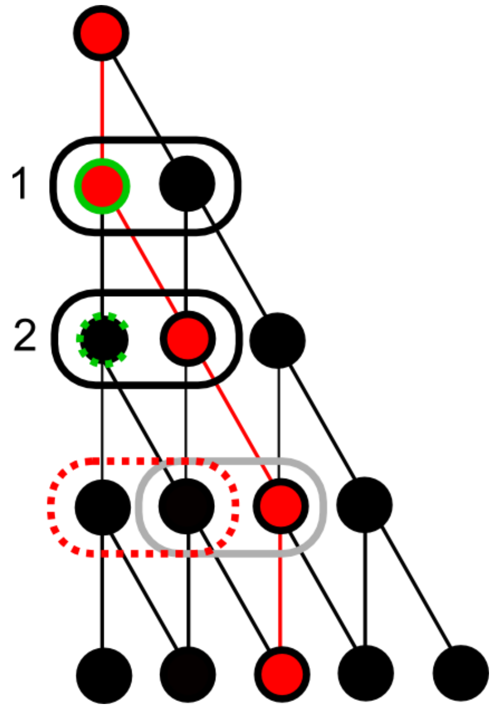
Create all possible sums (exhaustive search) best possible precision

Construct the sum from maxima of each harmonic. That is $h(n)_H = \sum_{i=1}^{H} \max_{1 \le j \le i} \left(x(in + j) \right)$

Greedy algorithm which selects highest value

$$h(n)_{H+1} = h(n)_H + \max(x(Hn + j), x(Hn + j + 1))$$

Harmonic sum – Tree view



There are few possibilities how to do harmonic sum

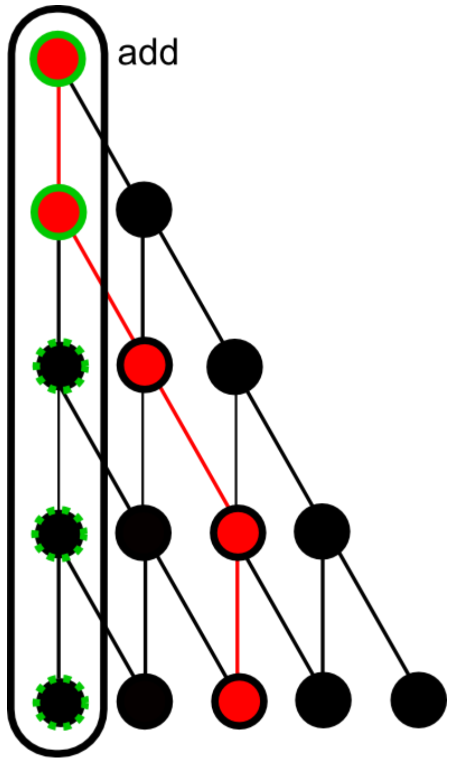
Create all possible sums (exhaustive search) best possible precision

Construct the sum from maxima of each harmonic. That is $h(n)_H = \sum_{i=1}^{H} \max_{1 \le j \le i}\left(x(in+j)\right)$

Greedy algorithm which selects highest value

$$h(n)_{H+1} = h(n)_H + \max(x(Hn+j), x(Hn+j+1))$$

Harmonic sum – Tree view



There are few possibilities how to do harmonic sum

Create all possible sums (exhaustive search) best possible precision

Construct the sum from maxima of each harmonic. That is $h(n)_H = \sum_{i=1}^{H} \max_{1\le j\le i}\left(x(in+j)\right)$

Greedy algorithm which selects highest value
$h(n)_{H+1} = h(n)_H + \max(x(Hn+j), x(Hn+j+1))$

Sum only integer multiples of the fundamental

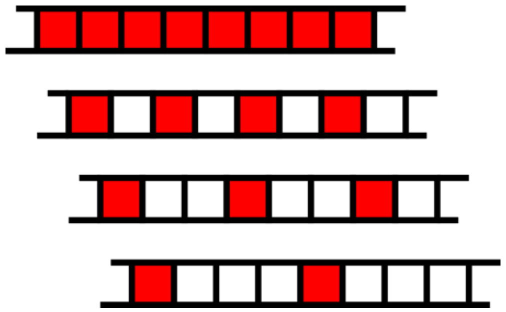
$$h(n)_H = \sum_{i=1}^{H} x(in)$$

Harmonic sum – Simple

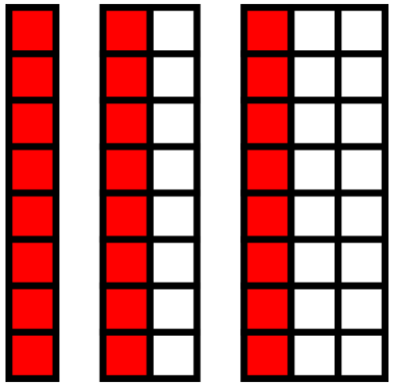
- Simple harmonic sum
  - Data are transposed, which improve data access
  - Device mem. bandwidth limited (77%)
  - No explicit data reuse caching is poor (10% L2 hit rate)

```
for(int f=1; f<nHarmonics; f++){
    pos = ...;
    partial_sum = partial_sum + data[pos];
    temp_SNR = (partial_sum - mean)/(stdev);
    if(temp_SNR > maxSNR) maxSNR = temp_SNR;
}
```

Memory access pattern when threads process neighboring fundamental frequency bins.



Memory access pattern when threads process same fundamental frequency bin from different data.



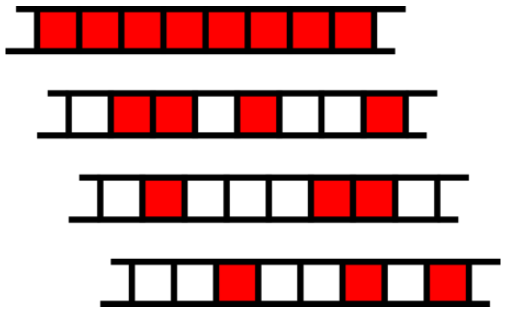
Harmonic sum – Greedy

- Greedy harmonic sum

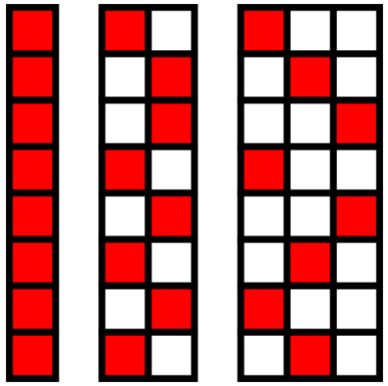
  For data reuse we really on caches (52% L2 hit rate)
  – kernel is waiting for data
  Device memory Bandwidth utilization (66%)

```
for(int h=1; h<nHarmonics; h++){
    int pos = ... + data_shift;
    data_down = d_input[pos];
    data_step = d_input[pos + 1];

    if(data_down<data_step) {
        partial_sum = partial_sum + data_step;
        data_shift++;
    }
    else {
        partial_sum = partial_sum + data_down;
    }

    temp_SNR = (partial_sum - mean)/(stdev);
    if(temp_SNR>maxSNR) maxSNR = temp_SNR;
}
```

Memory access pattern when threads process neighboring fundamental frequency bins.



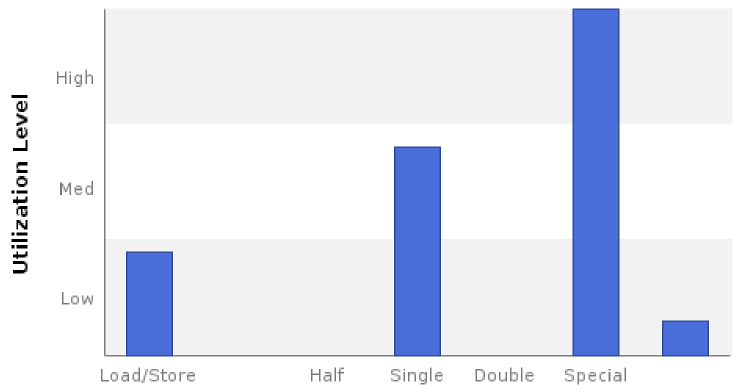
Memory access pattern when threads process same fundamental frequency bin from different data.



Harmonic sum – Presto, MaxDIT

Presto harmonic sum

Limited by type conversion (array indexing). However fp32 compute is still high.

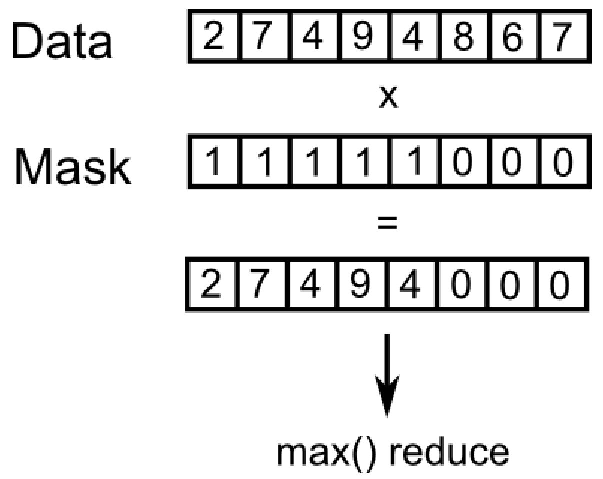


Max harmonic sum

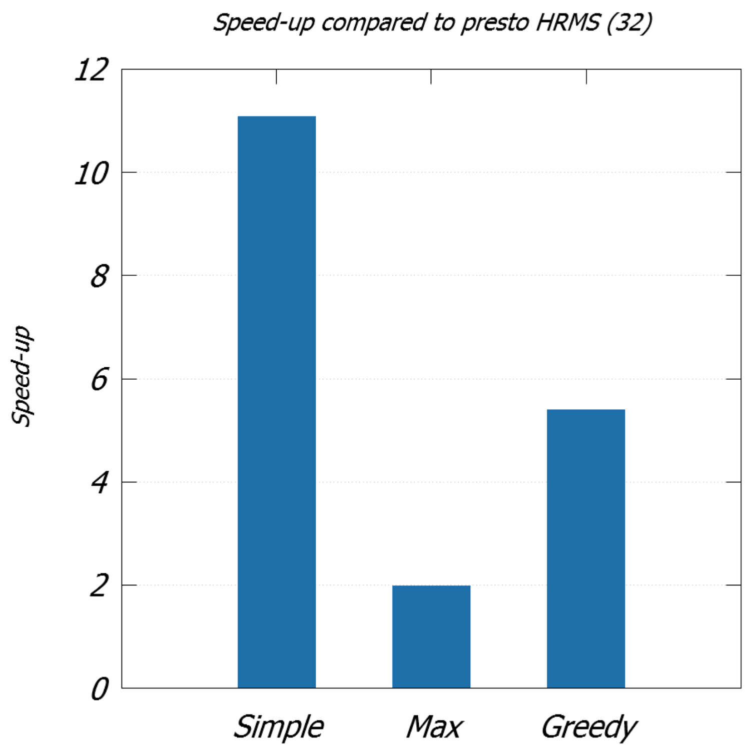
Max harmonic sum is two step

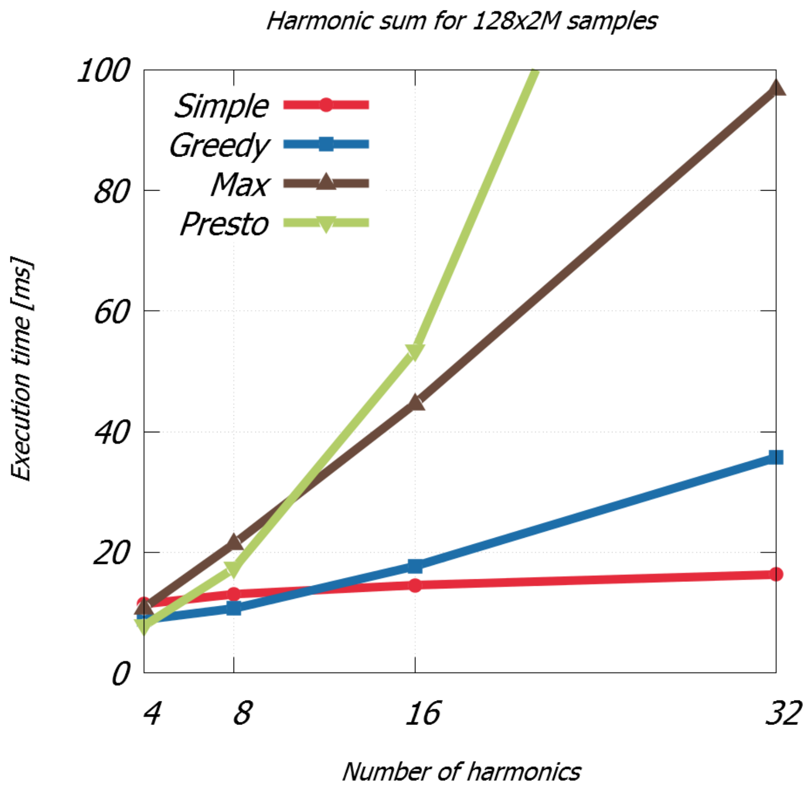
1) Calculate all max decimations - Limited by compute (Load/Store, floating point operations)

2) Calculate partial sums and SNR



Harmonic sum – Improvements

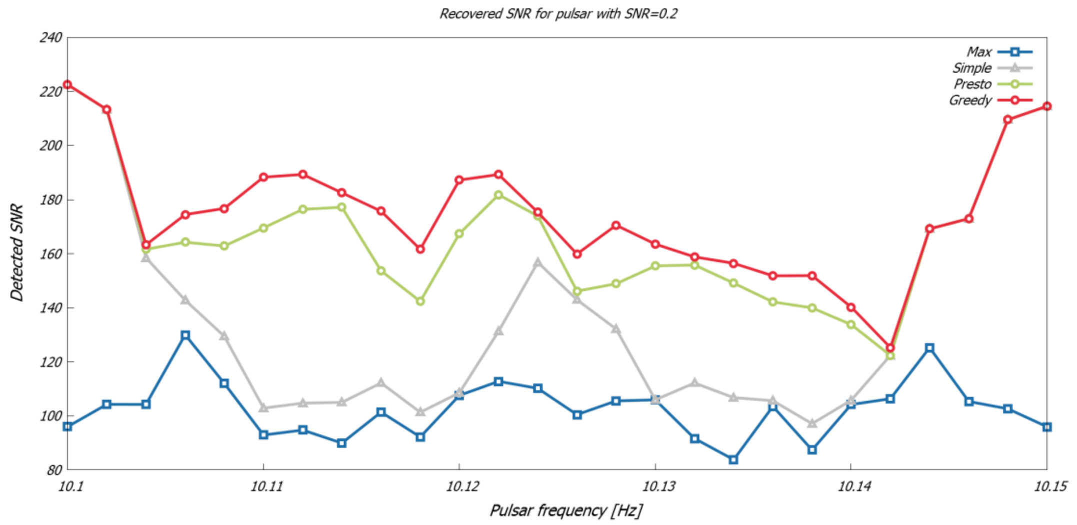


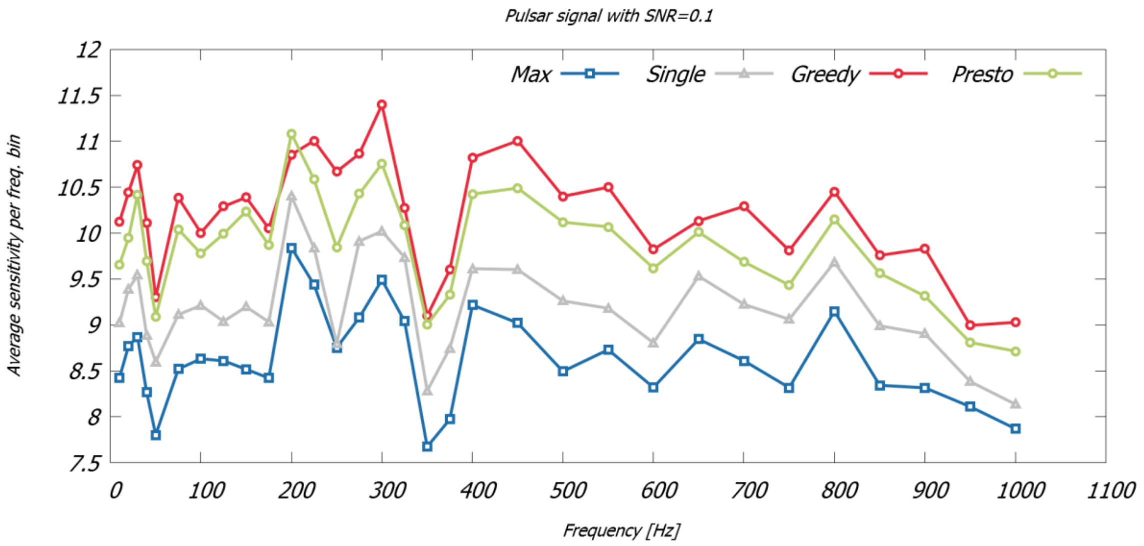


Harmonic sum – Results

As gold standard we have used our implementation of presto‘s HRMS.

RIGHT: Sensitivity loss for pulsar frequencies which are between frequency bins. 50% decrease for simple HRMS, only 30% for Greedy and PRESTO.





LEFT: average sensitivity as it depends on pulsar’s frequency.

Harmonic sum – Conclusions

- We have multiple algorithms with different parameters
- We need more sensitivity tests and tests of physical correctness (artificial data, real data)
- We thinking about 2D harmonic sum for acceleration searches
- We trying to increase performance