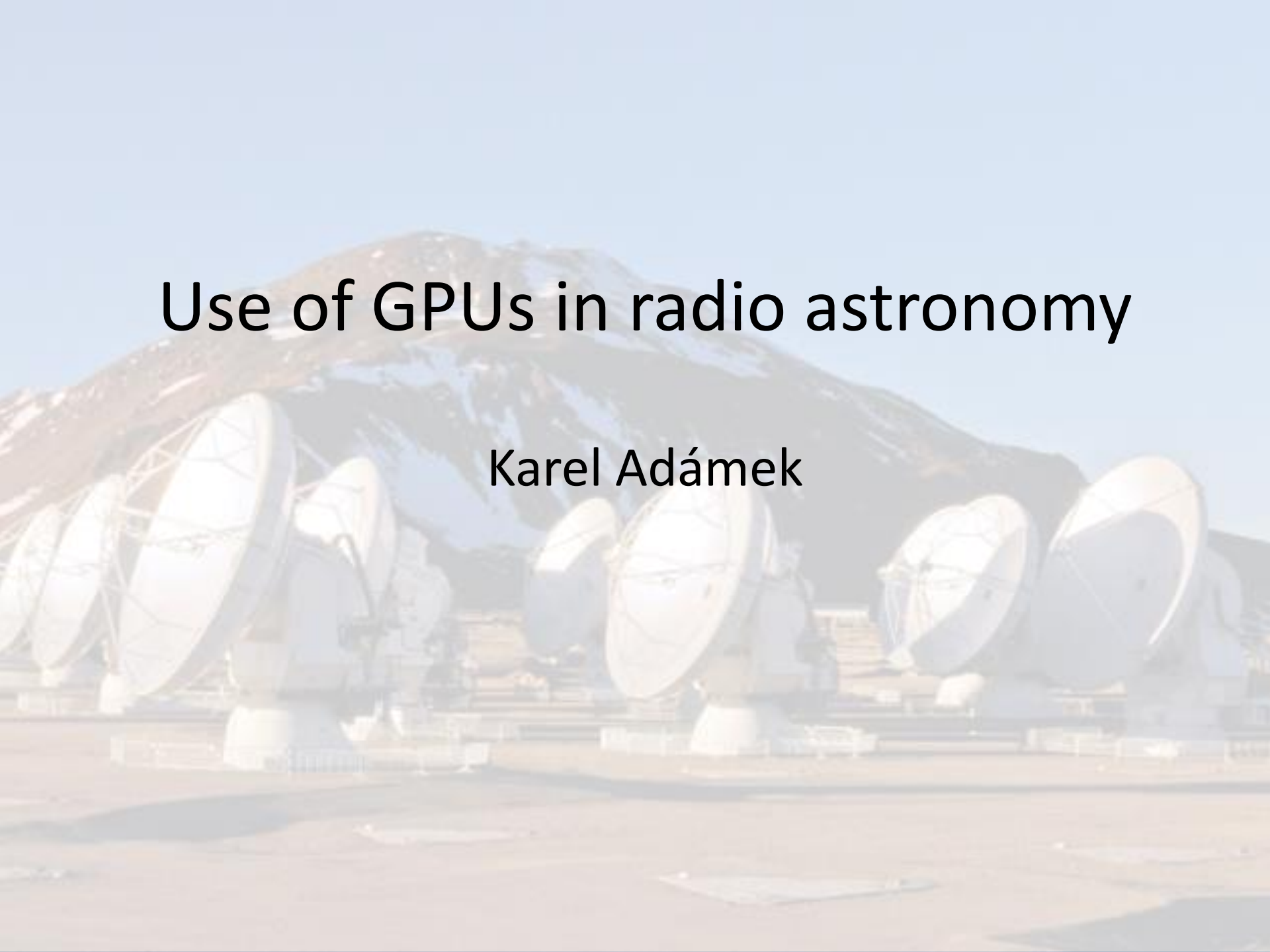
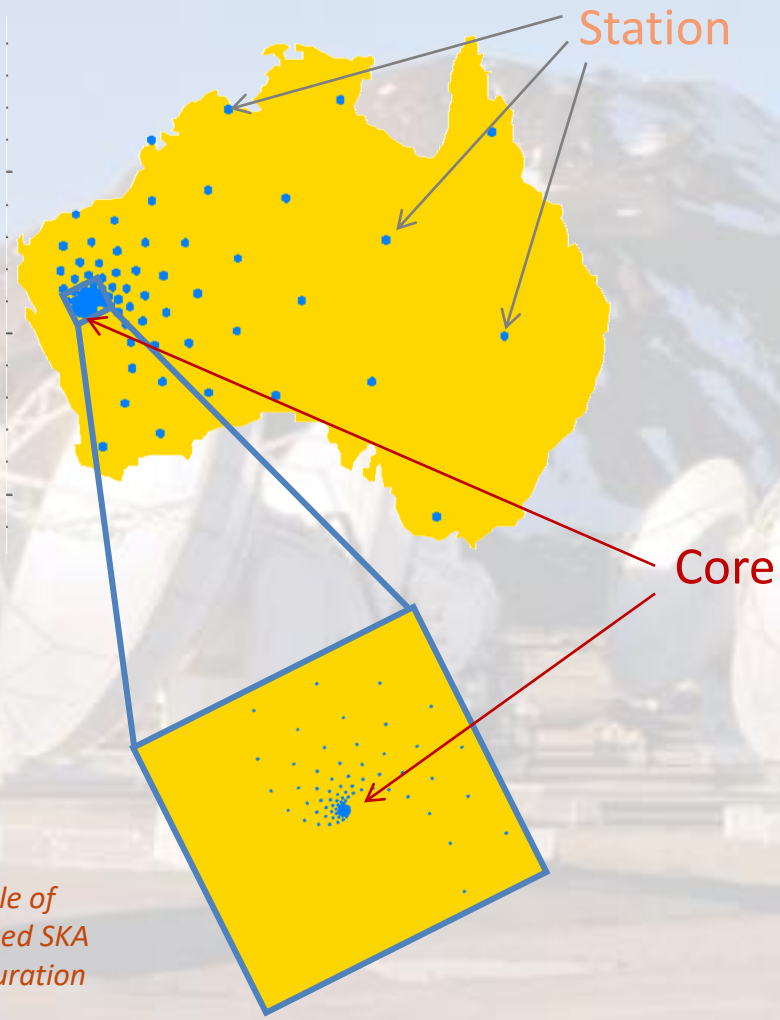


Use of GPUs in radio astronomy

Karel Adámek

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

Radio astronomy, its data and operations

Typical operations in radio astronomy are low in arithmetic intensity (number of floating point operations per byte) =>

Data locality and reuse are critical for performance

Data are separable into independent chunks (mostly)
In time-domain it is 10min observation (6GB) which needs to be processed faster than real-time.

SKA	Time-domain	Image-domain
Data-rate	~160 GB/s	0.5—1 TB/s
Data	Multiple time-series	Multiple 2D freq-domain data
Size of data product	Small	32k×32k×64k image cube ~ 64 TB
Product domain	Events	Image-domain

AstroAccelerate

AstroAccelerate is an open source GPU accelerated software package for processing time-domain radio astronomy data for Square Kilometre Array (SKA).

It is currently used at:

- Giant Metrewave Radio Telescope (India)
- MeerKAT (South Africa; SKA path finder)
- PetaScale project (West Virginia University) – reprocessing of 0.5PB of archival data

AstroAccelerate team was part of time-domain team within SKA.

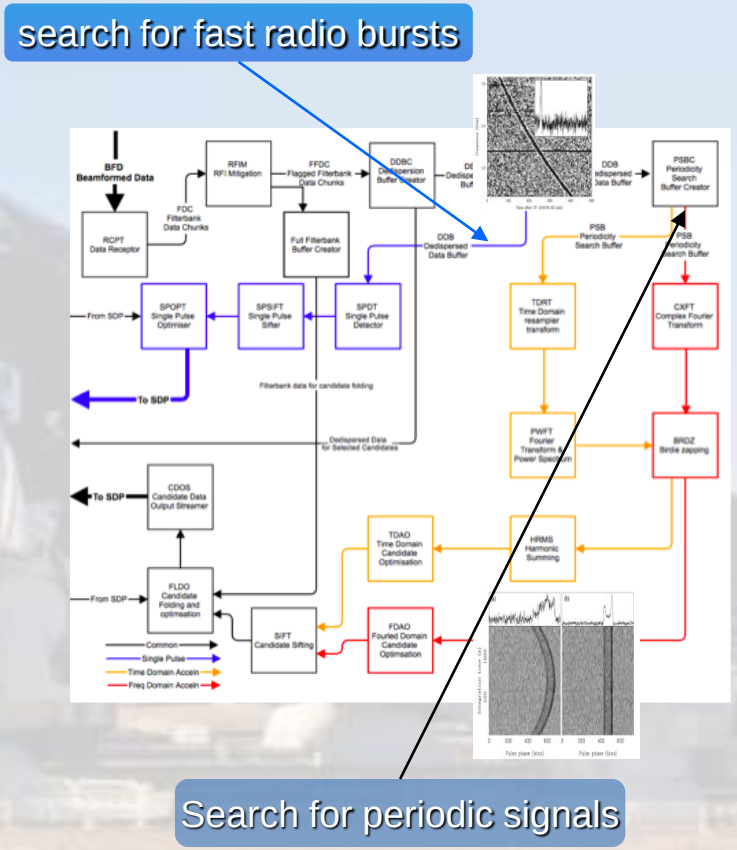
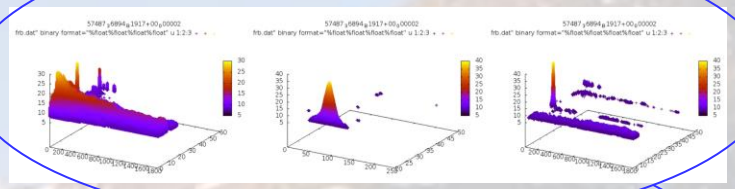


Image courtesy of Aris Karastergiou

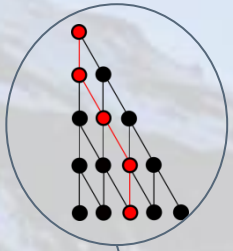
Time Domain Team

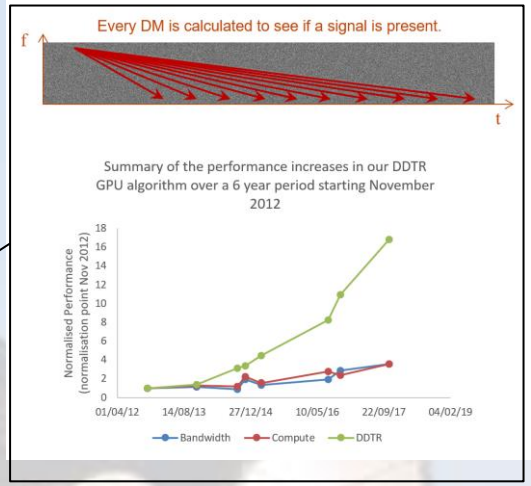
AstroAccelerate - Signal Processing

Radio Frequency Interference Mitigation



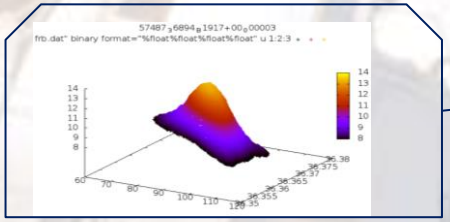
Harmonic Sum

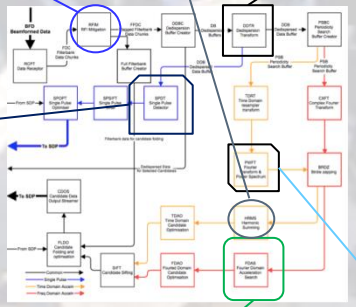




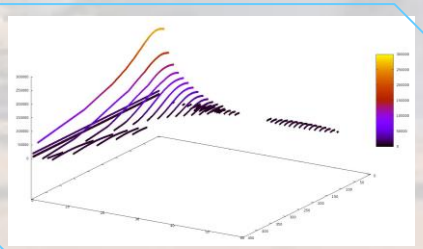
De-dispersion

Single Pulse Search

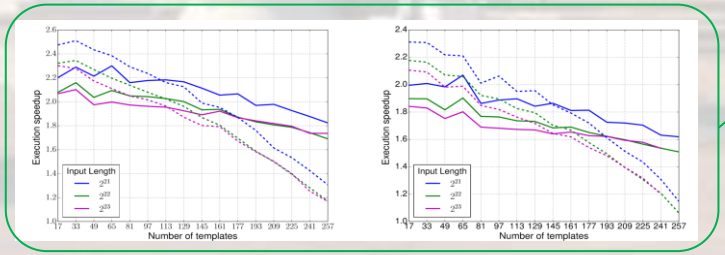




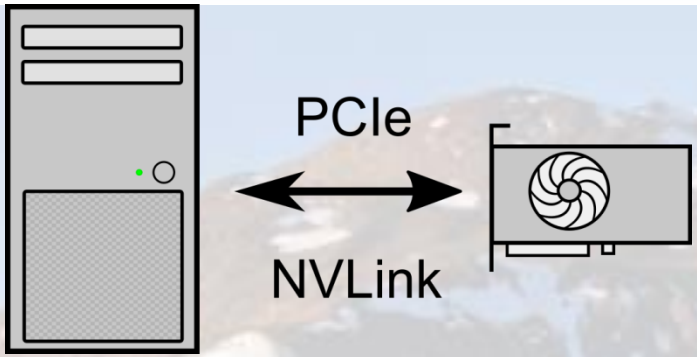
Periodicity Search



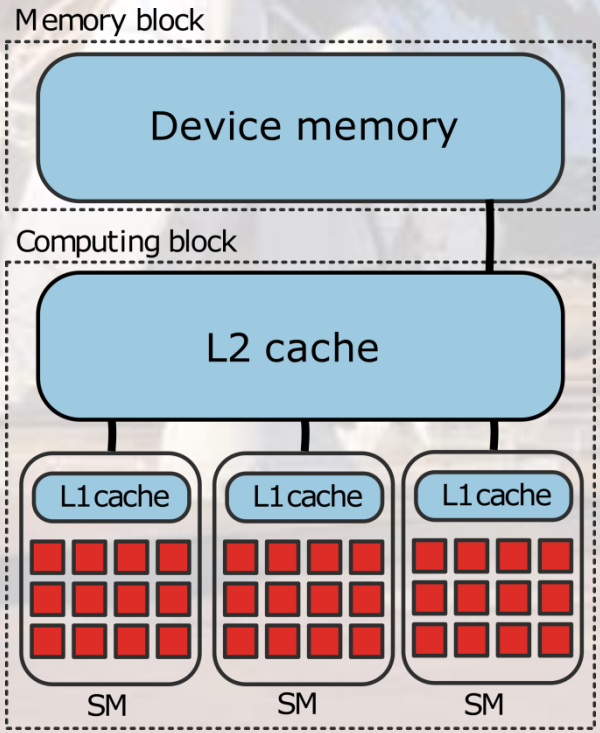
Fourier Domain Acceleration search



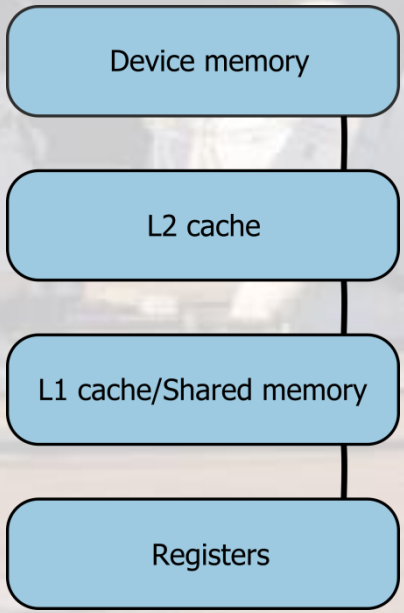
GPU - Architecture



Bandwidths:
CPU bandwidth: ~300GB/s
GPU bandwidth: ~1500GB/s (A100)
PCIe bandwidth: ~32GB/s (v4)
NVLink bandwidth: ~300GB/s (3.0 on A100)

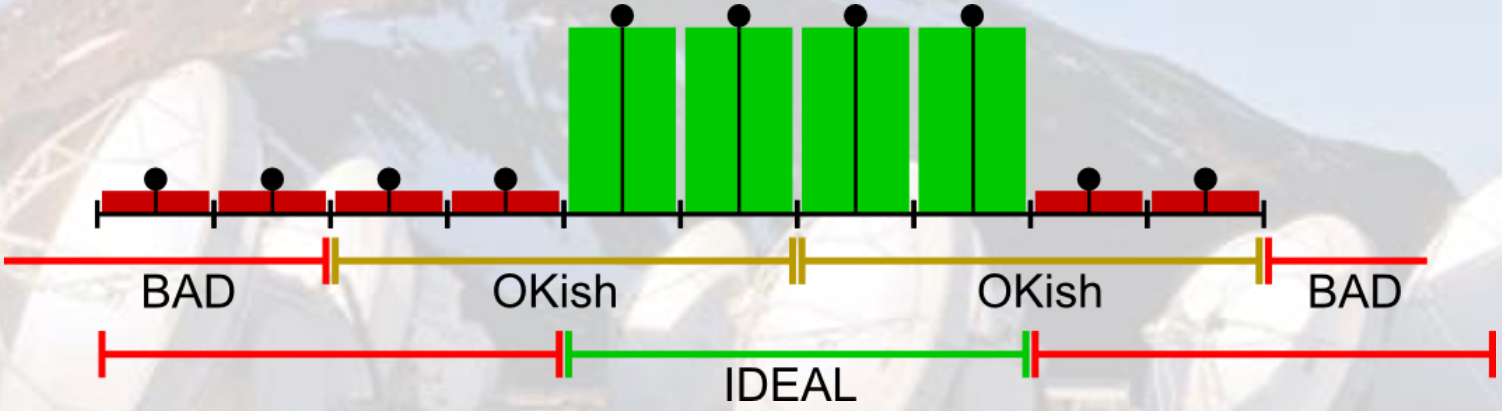


GPU memory hierarchy



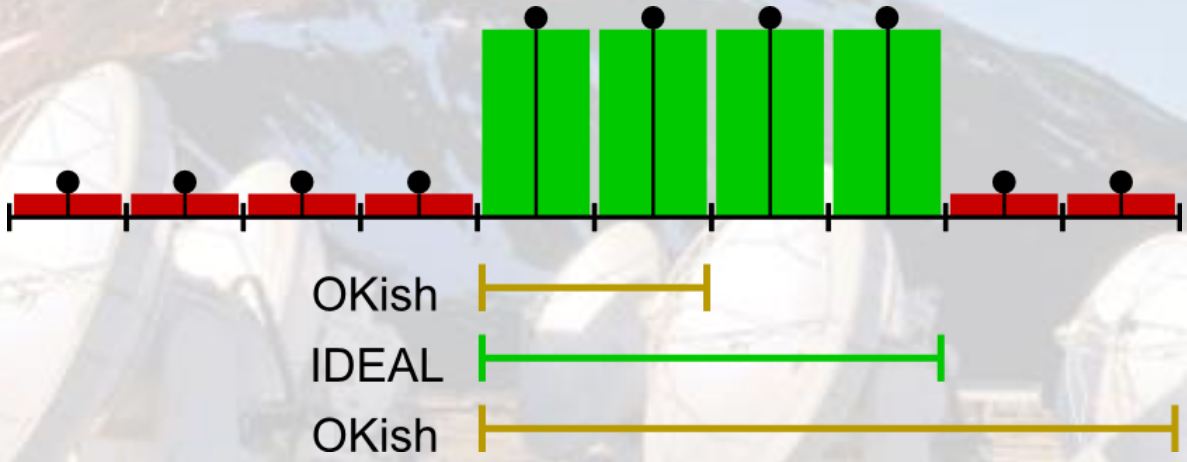
Time-domain: Single pulse detection

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



Time-domain: Single pulse detection

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



Time-domain: Single pulse detection

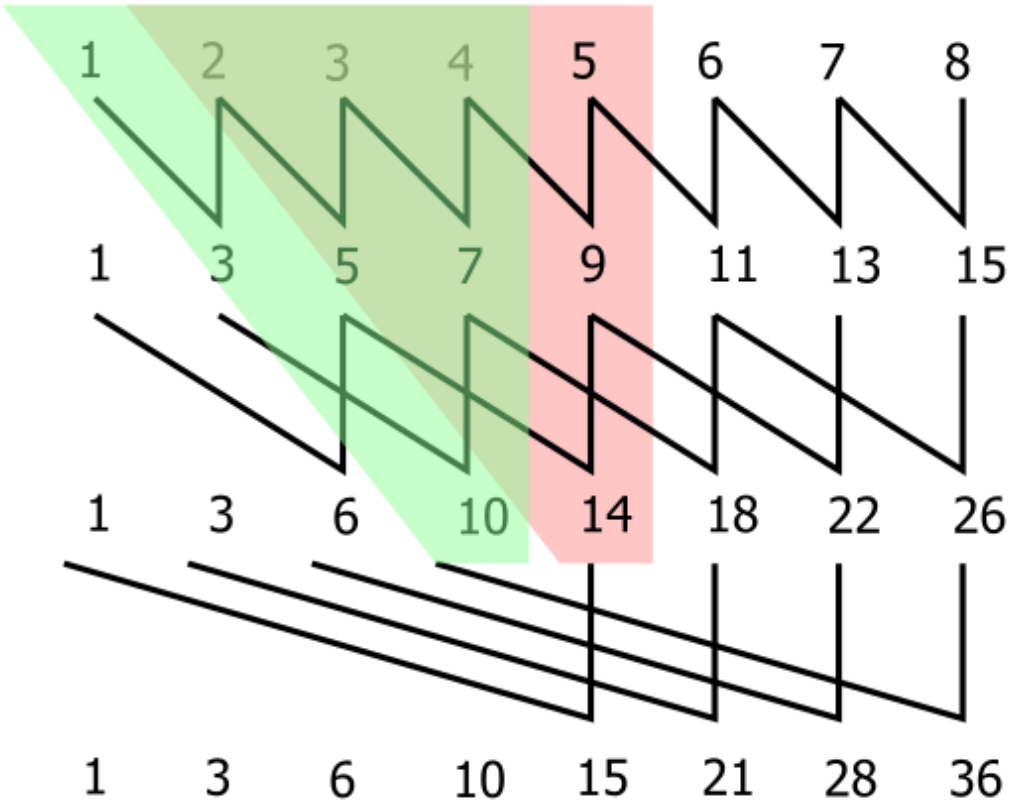
Aim is to detect pulses of different shapes and widths at unknown position within the signal and do it quickly.

- Position of the boxcar is important
- Length of the boxcar is important

→

Prefixed sum (scan) of length N must be computed at each sample of the time-series and select highest SNR for each sample.

Time-domain: Scan



Hillis Steele Scan

Is fast but for our needs did not reused data enough.

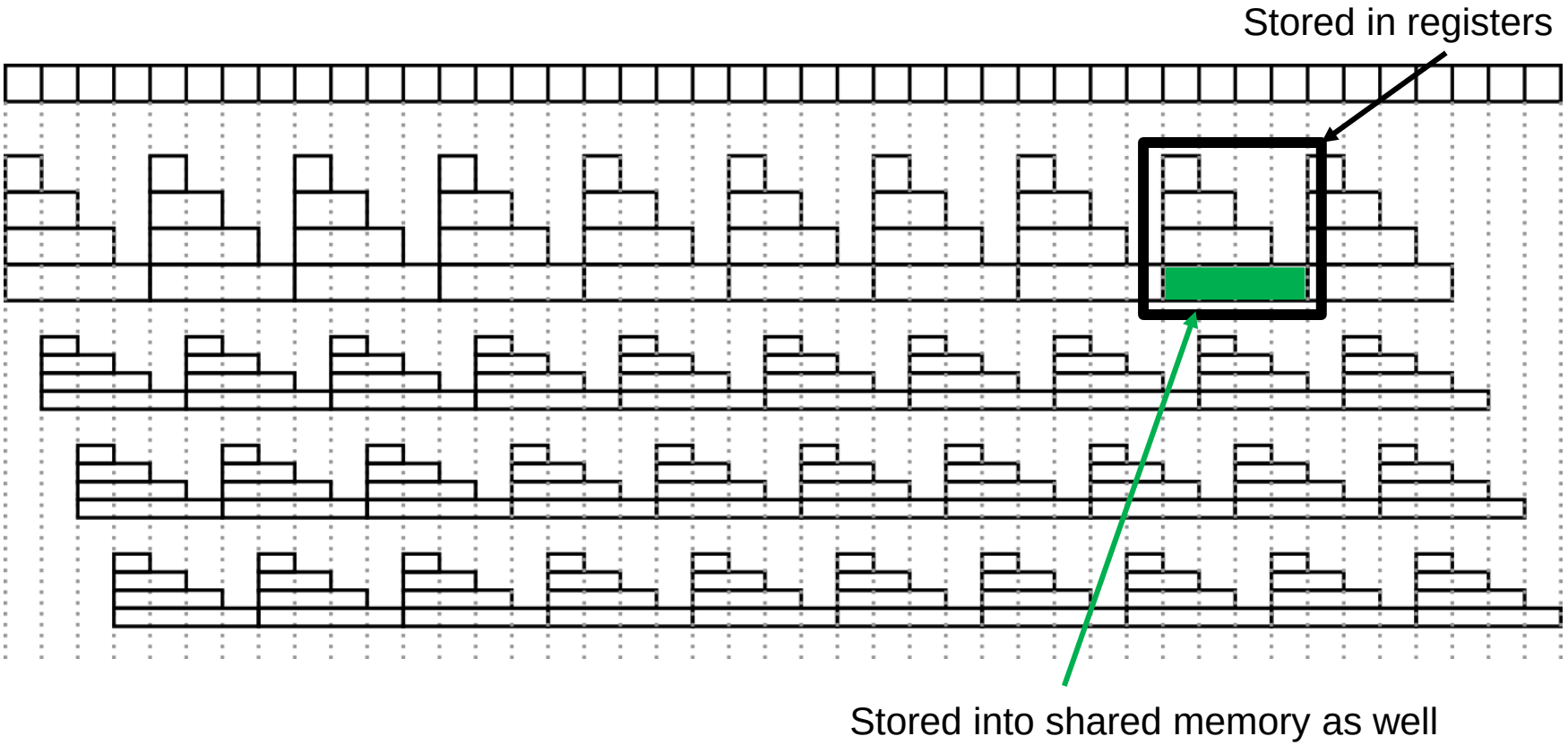
Green area represent part of the scan starting at number 1.

Red area represent part of the scan starting at number 2.

For higher performance we have to increase data reuse. To do that we must increase the number of scans calculated per thread.

Time-domain: Scan – more data reuse

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.



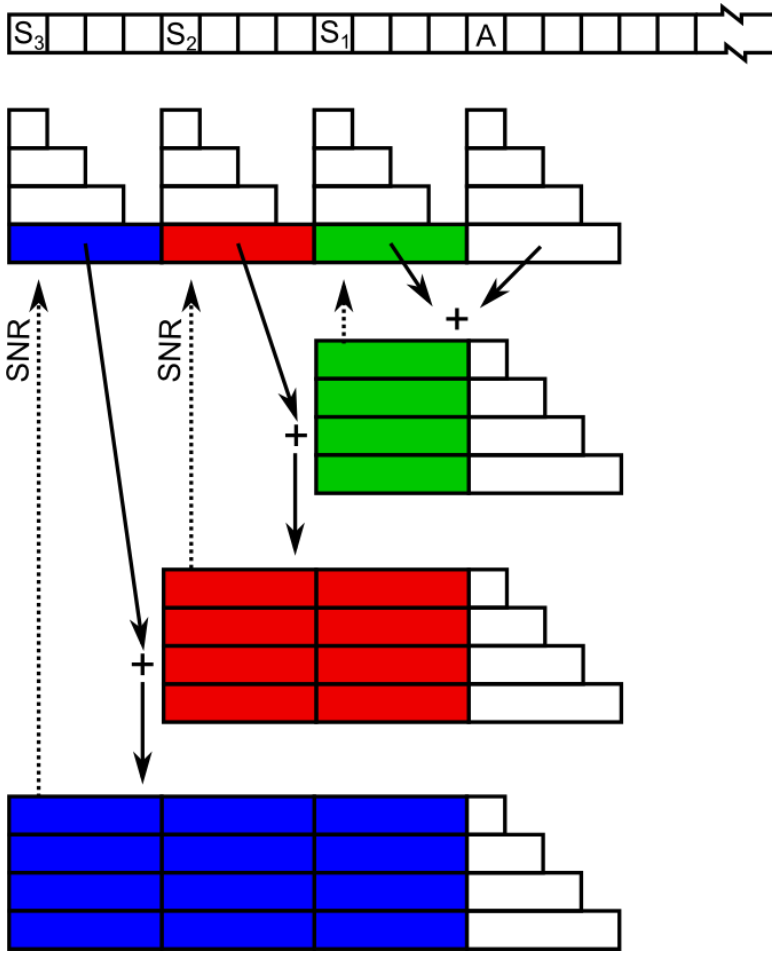
Time-domain: Single pulse detection

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.

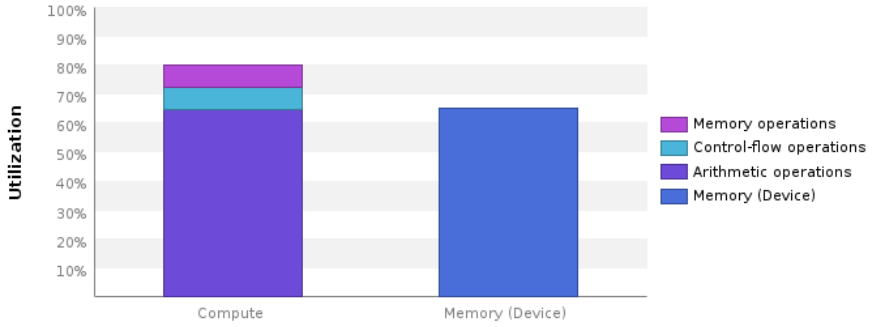
In each step i, an active thread A, calculates a source thread id as $S_i = A - i*4$. The value of the longest boxcar calculated at beginning is loaded from source thread (from shared memory) and used to calculate longer boxcars in the active thread. These are kept by the active thread.

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.



Time-domain: Single pulse detection

We have turned a device memory bandwidth limited algorithm into compute limited algorithm.



Other versions of BoxDIT algorithm

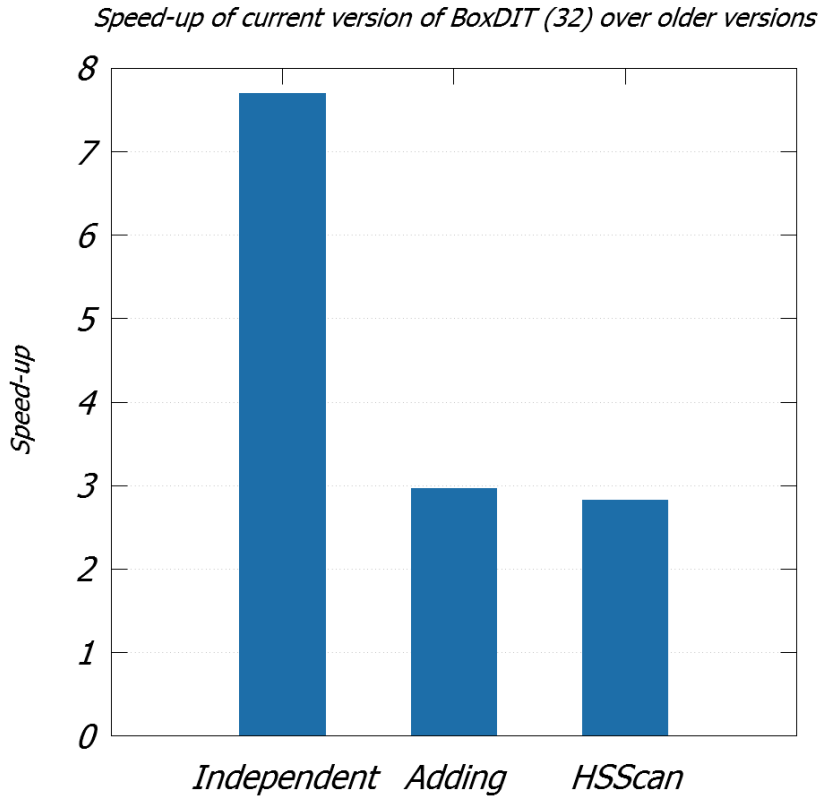


Image-domain: Crash course in radio imaging

Visibilities (points in freq-domain) are gathered by a telescope. Freq-domain is not sampled regularly => cannot use FFT and imaging process is iterative to reconstruct the true image of the sky.

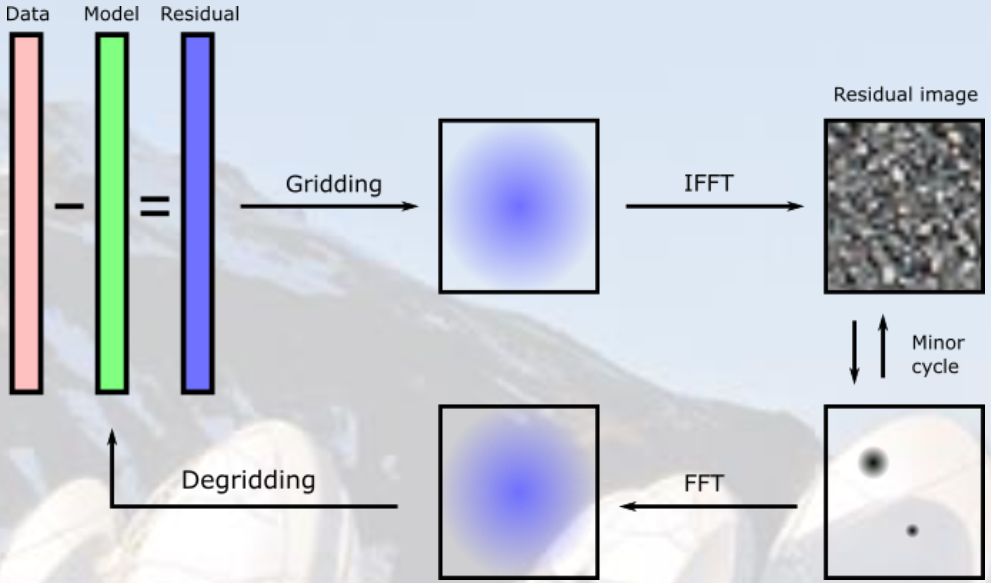
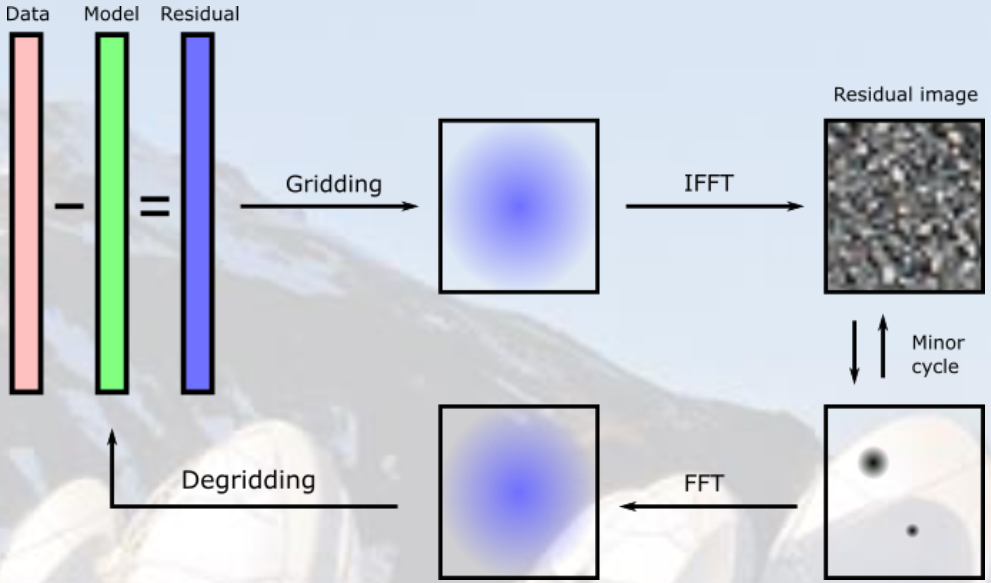
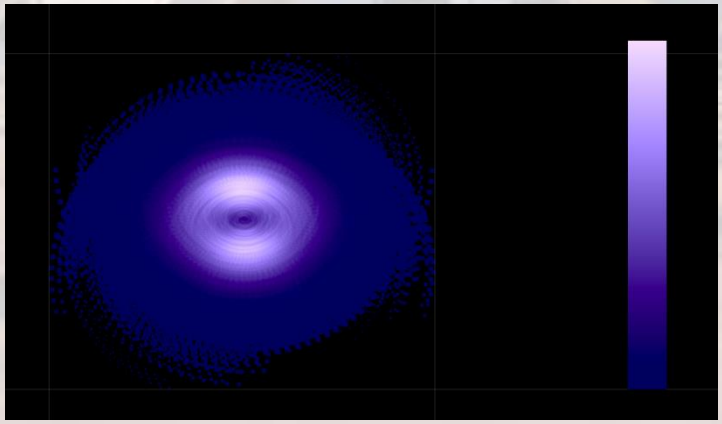


Image-domain: Crash course in radio imaging

Visibilities (points in freq-domain) are gathered by a telescope. Freq-domain is not sampled regularly => cannot use FFT and imaging process is iterative to reconstruct the true image of the sky.



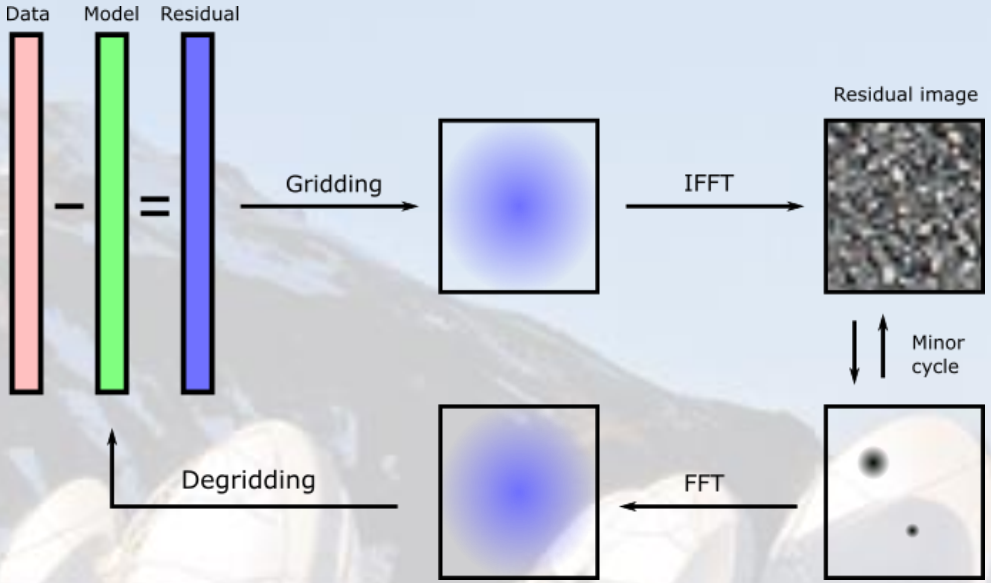


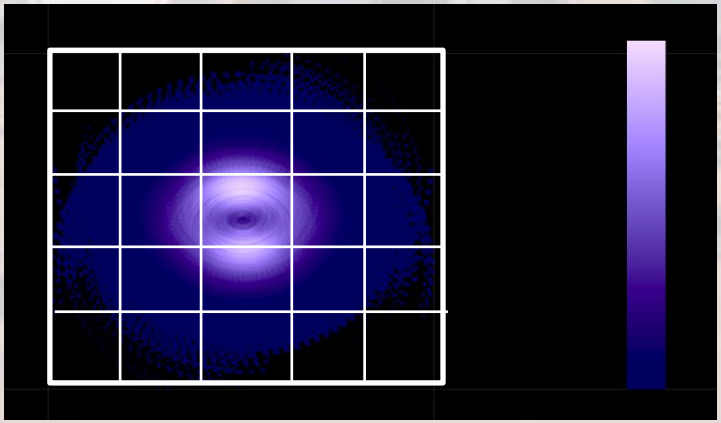
The visibilities are not distributed evenly. This represent load balancing problem for image processing pipelines.

The visibilities are regularized before use by gridding algorithm and de-regularized by degridding algorithm.

Image-domain: Crash course in radio imaging

Visibilities (points in freq-domain) are gathered by a telescope. Freq-domain is not sampled regularly => cannot use FFT and imaging process is iterative to reconstruct the true image of the sky.

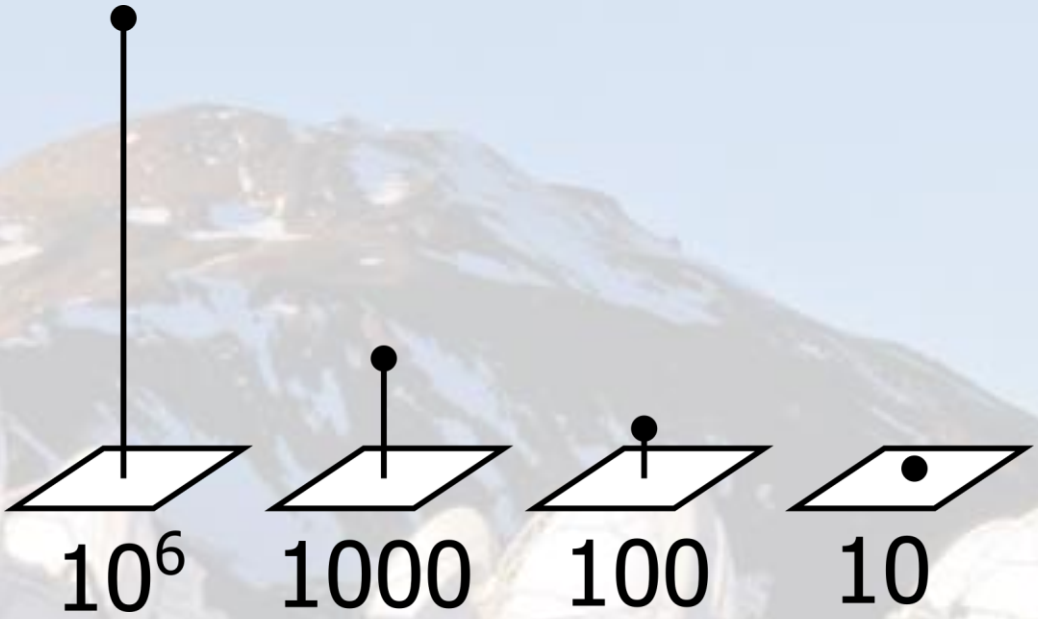




The visibilities are not distributed evenly. This represent load balancing problem for image processing pipelines.

The visibilities are regularized before use by gridding algorithm and de-regularized by degridding algorithm.

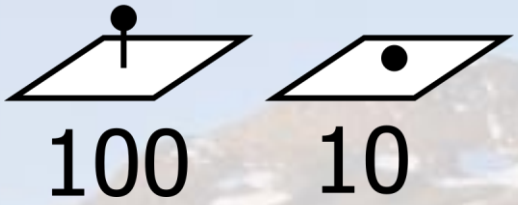
Uneven distribution of visibilities

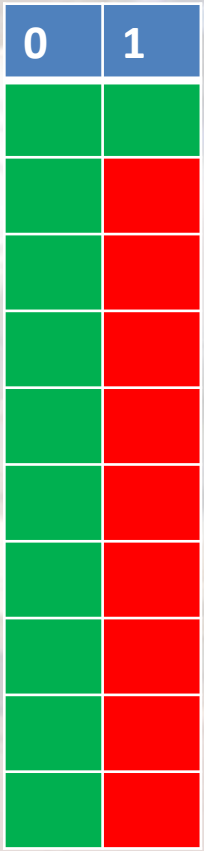


Where uneven distribution of visibilities can cause problems:

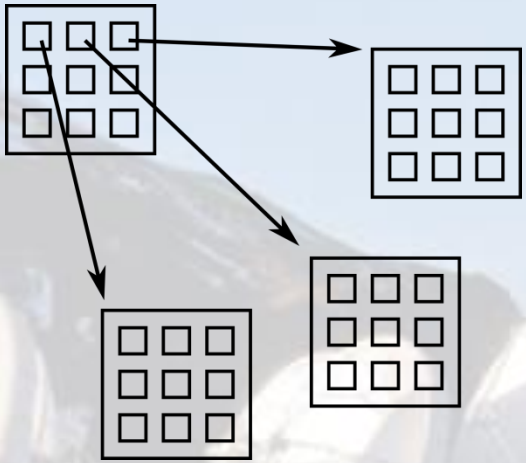
- GPU kernel itself where we need dynamic for loop to go through visibilities
  - Not a problem as scheduling on GPUs is getting better; performance drop is marginal
- Configuration of the GPU grid
  - Grid is rigid for given GPU kernel execution
  - Multiple executions are possible but introduce overheads

What is the problem





Can dynamic parallelism help?



- Allows a thread to spawn its own CUDA grid which can fit the needs of given subgrid
- Works OKish for large differences in number of visibilities
- Induce large overheads which for similar number of visibilities reduces performance

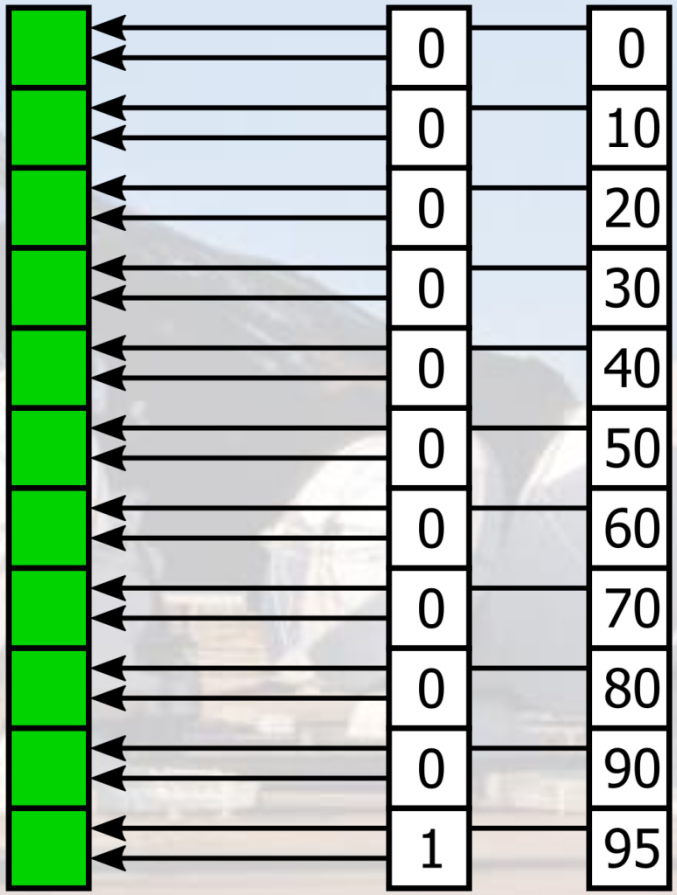
Solution: flatten the CUDA grid

Instead of allocation of 2D CUDA grid we have used 1D CUDA grid where work is allocated using two prefixed sums

First prefix sum contains subgrid index thus eliminating inefficiencies inherent for 2D CUDA grid

Second prefixed sum contains starting visibility.

Performance increase (depending on subgrid configuration) is: 0% -- 15%



karel.adamek@gmail.com