Lecture 8 - AstroAccelerate

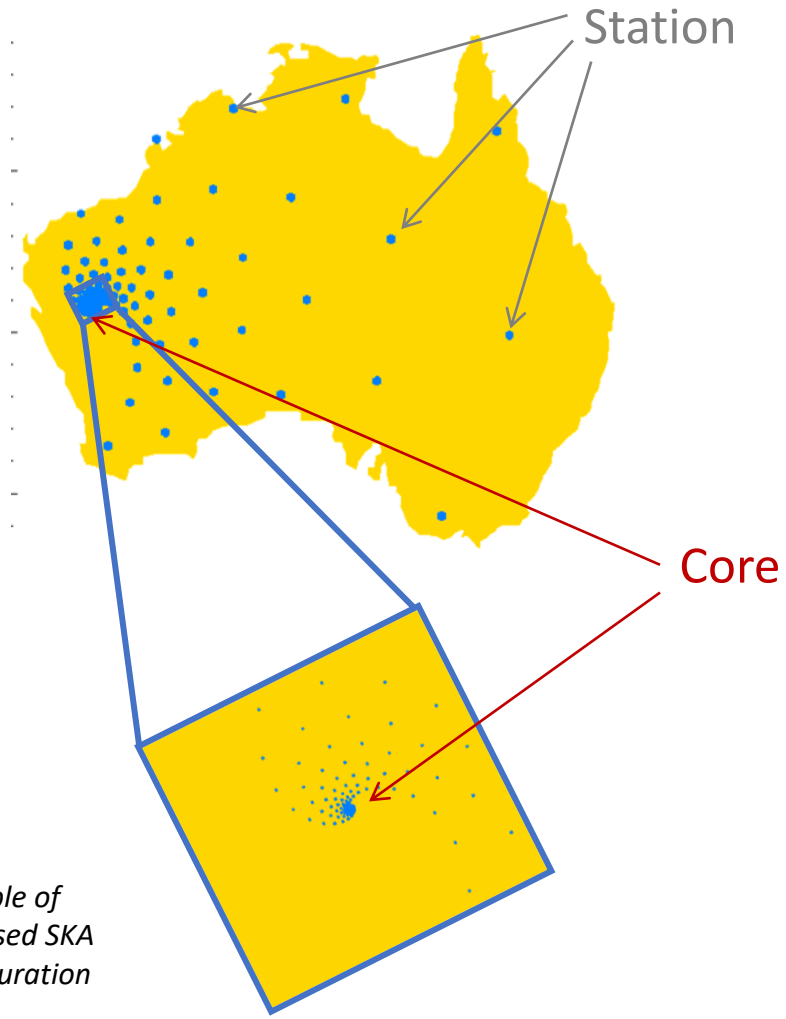
GPU accelerated signal processing for next generation radio telescopes

Wes Armour, Karel Adamek, Cees Carels, Kate Clark, Sofia Dimoudi, Mike Giles, Jan Novotny, Nassim Ouannough, Scott Ransom, Jayanta Roy, Jack White

Part One

A brief introduction to

What is SKA?



Example of proposed SKA configuration

What is SKA?

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

What does SKA stand for?

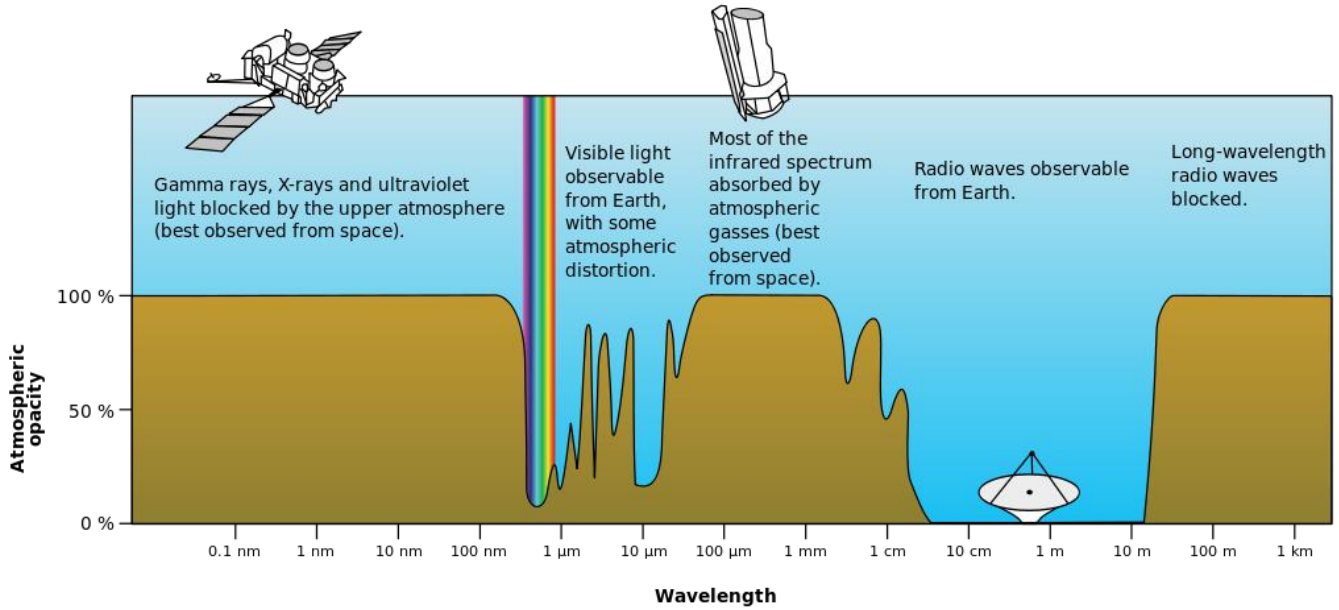
*Square Kilometre Array, so called because it will have an **effective collecting area of a square kilometre**.*

Where will SKA be located?

*SKA will be built in **South Africa and Australia.***

Graphic courtesy of Anne Trefethen

What is SKA?



SKA is a ground based telescope. This means that it is most sensitive to the radio range of frequencies. The radio range of frequencies that can be observed from here on Earth is very wide, specifically SKA will be sensitive to frequencies in the range of 50MHz to 20GHz (**wavelengths 15 mm to 6 m**). This makes SKA ideal for **studying lots of different science cases.**

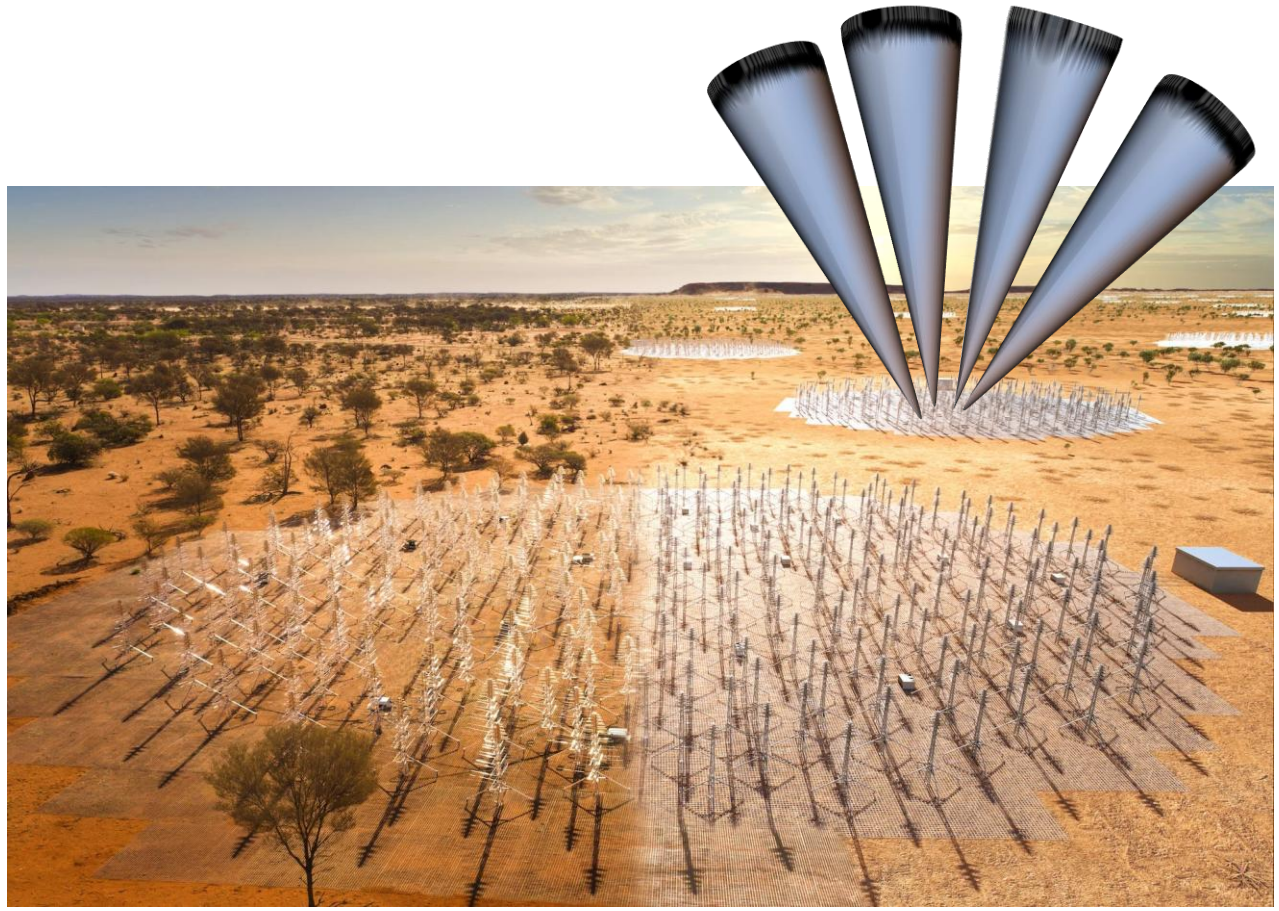
Image source Wikipedia. Authors: NASA (original); SVG by Mysid

What is SKA?

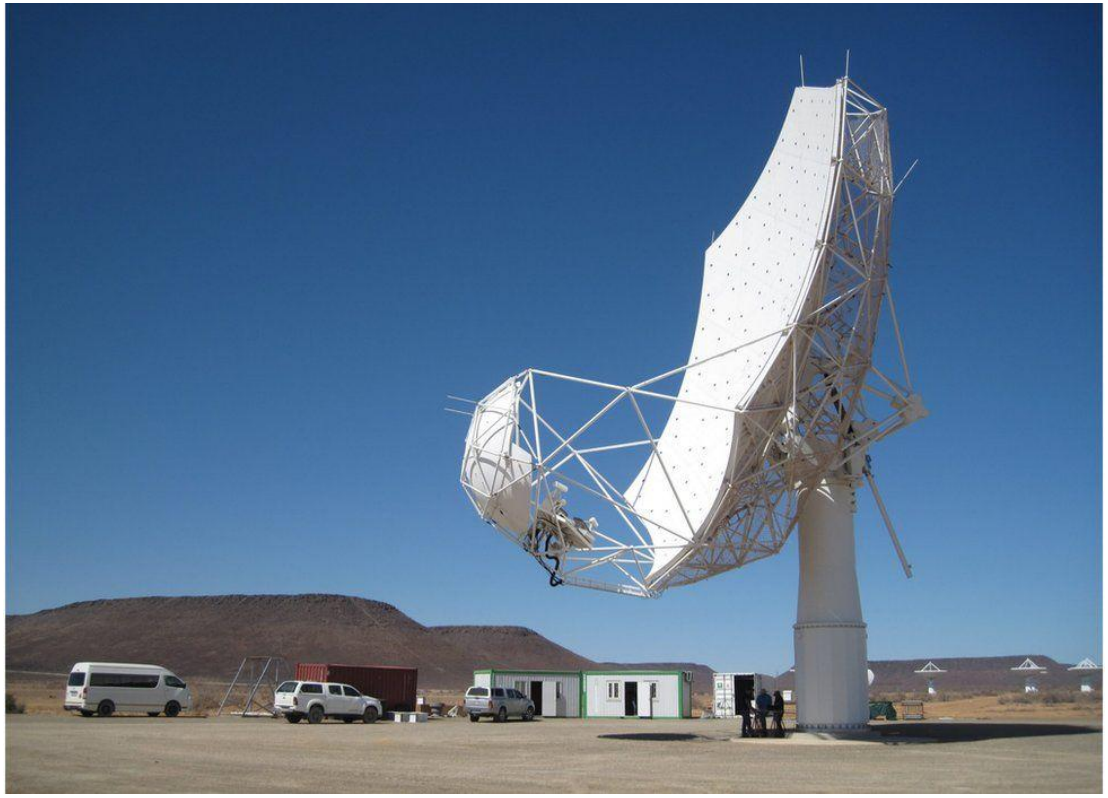
SKA will have the ability to use all of its antennas to **produce images of the radio sky in unprecedented accuracy and detail.**

It will also be able to use combinations of antennas to perform **multiple observations of different regions of the sky at the same time**.

In this scenario data from each beam can be computed in parallel.



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?

SKA time domain - signal processing

Time Domain Team

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).

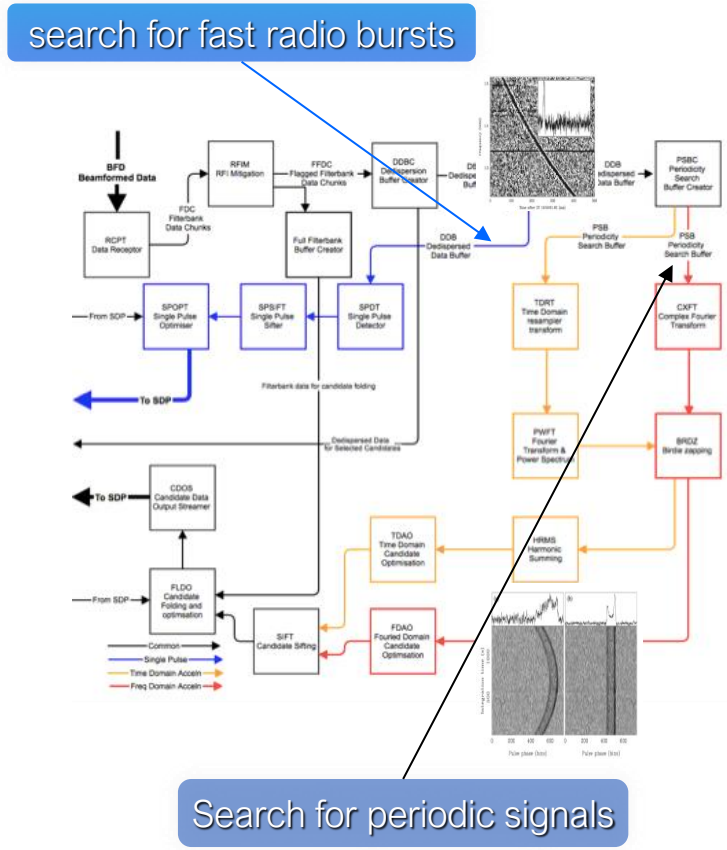


Image courtesy of Aris Karastergiou

SKA time domain - signal processing

Our work focused on vertical prototyping activities.

We delivered accelerated algorithms for 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.

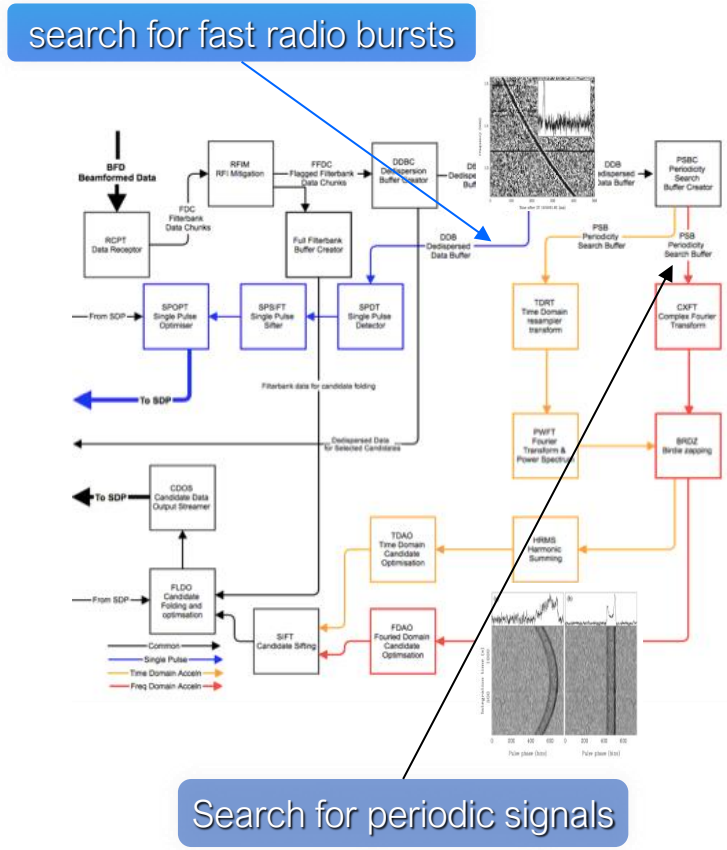


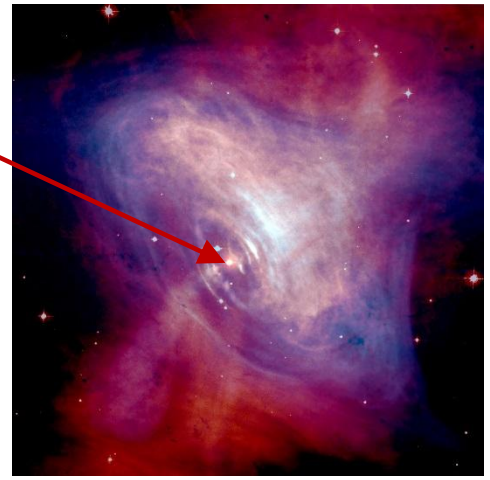
Image courtesy of Aris Karastergiou

SKA time domain science - Pulsars

Pulsars are magnetized, rotating neutron stars. They emit synchrotron radiation from the poles, e.g. Crab Nebula.

Their magnetic field is offset from the axis of rotation as such (as observed from here on Earth, they act as cosmic lighthouses.

They are extremely periodic and so make excellent clocks!



Hester et al.

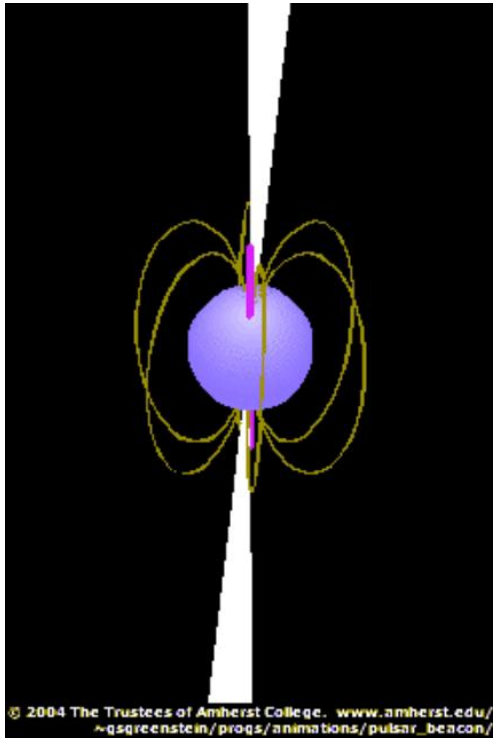
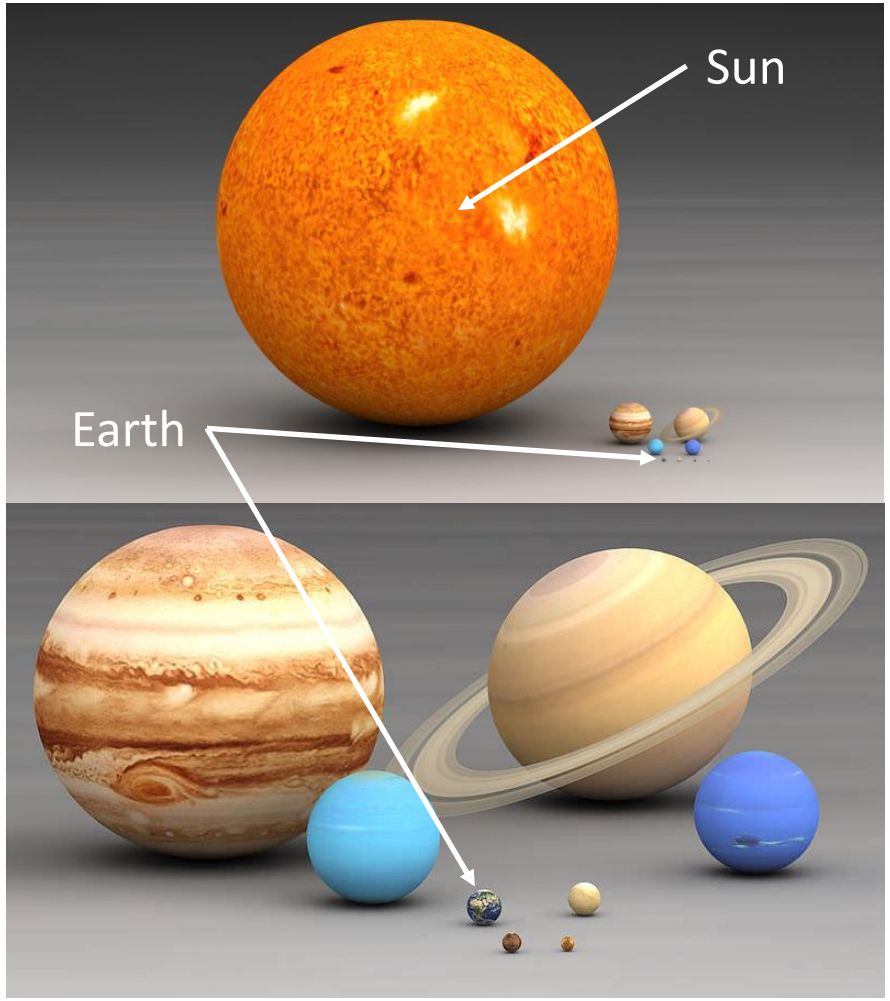


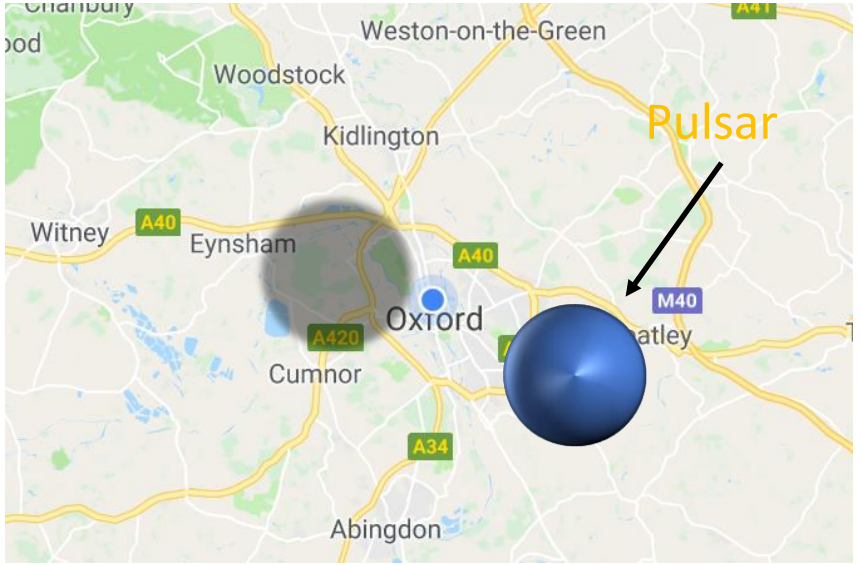
Image: Amherst College

Pulsars – size and scale



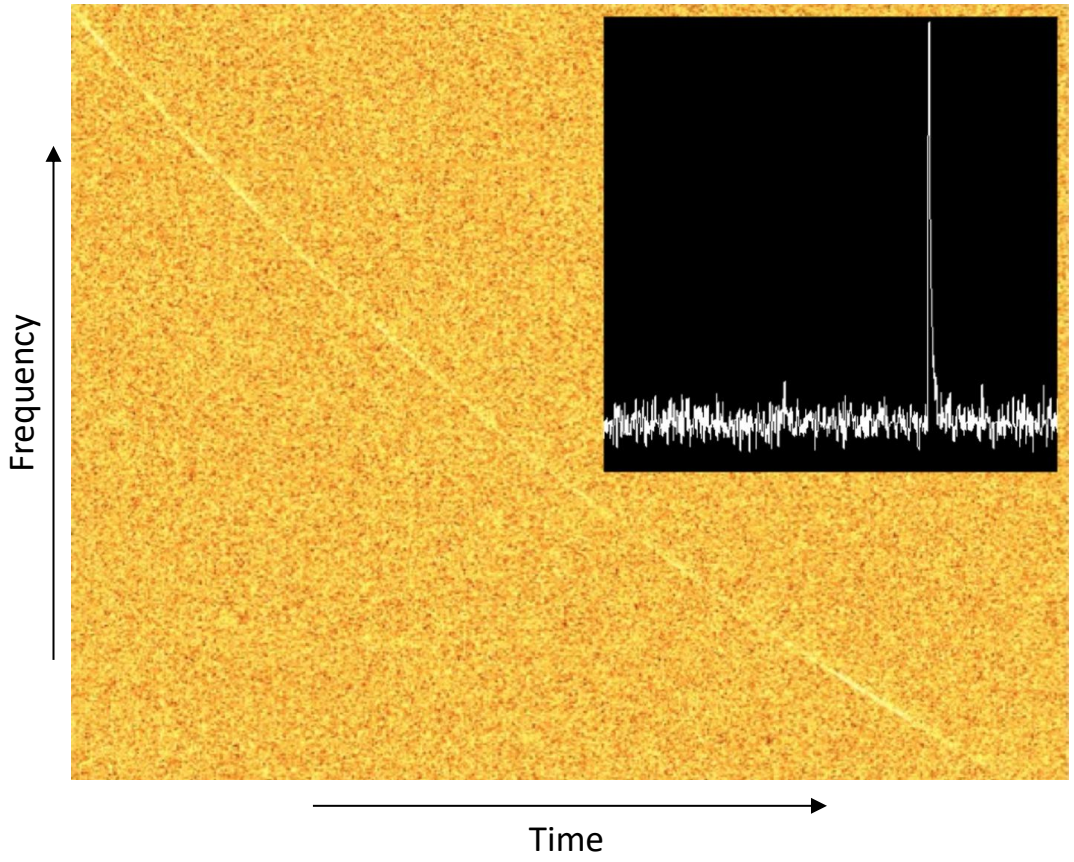
Pulsars are typically 1-3 Solar masses in size, they have a diameter of 10-20 Kilometres and a pulse period ranging from milliseconds to seconds.

Meaning that they are very small, very dense and rotate extremely quickly.



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

SKA time domain science - FRBs



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).

Now hundreds of FRBs have been observed or found 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)

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!

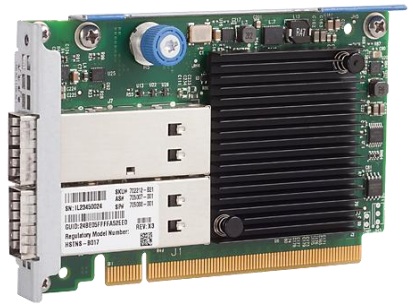
SKA time domain – data challenges

Because we would like to **monitor interesting and exotic events as they occur, we need to process data in real-time** (or as near to as possible).

So, storing the data and processing later isn't feasible. The data rates mean transporting data offsite would be challenging and costly.

Hence processing must happen close to the telescope. But how do we put a computer capable of processing big-data streams in real-time close to the telescope?

Connectivity, power, operation all pose significant problems.



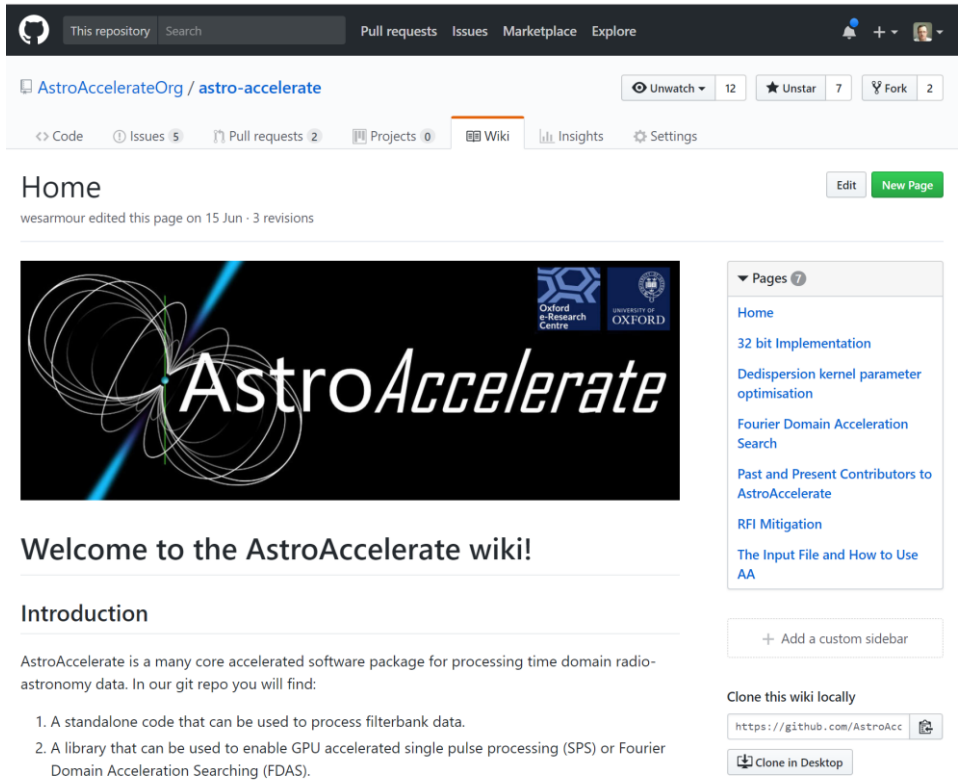
Part Two

Astro*Accelerate* – A case study

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.

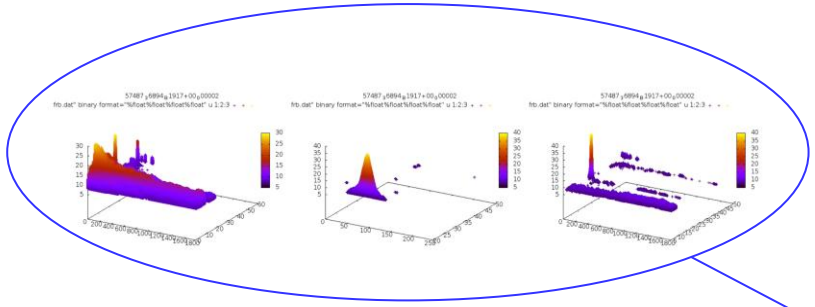
The massive computational power of modern-day GPUs allows the code to perform algorithms such as de-dispersion, single pulse searching and Fourier Domain Acceleration Searching in real-time on very large data-sets which are comparable to those which will be produced by next generation radio-telescopes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



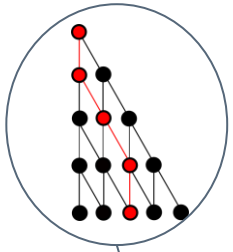
https://github.com/AstroAccelerateOrg/astro-accelerate

AstroAccelerate - Signal Processing

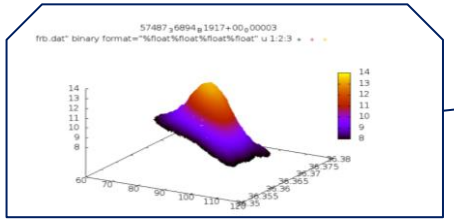
Radio Frequency Interference Mitigation

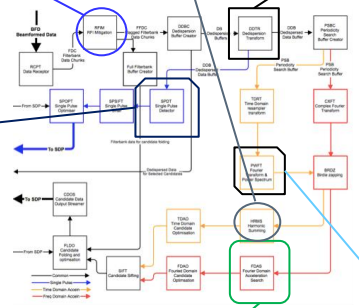


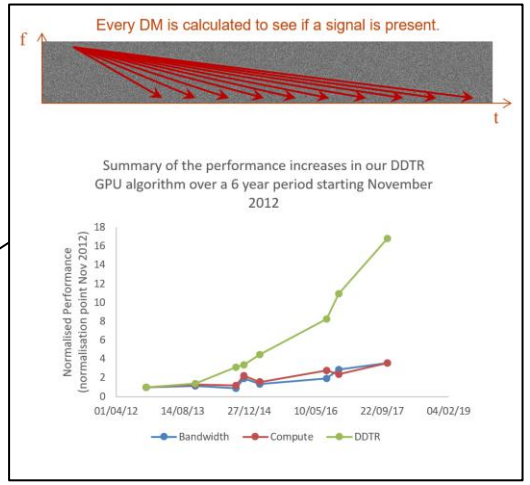
Harmonic Sum



Single Pulse Search

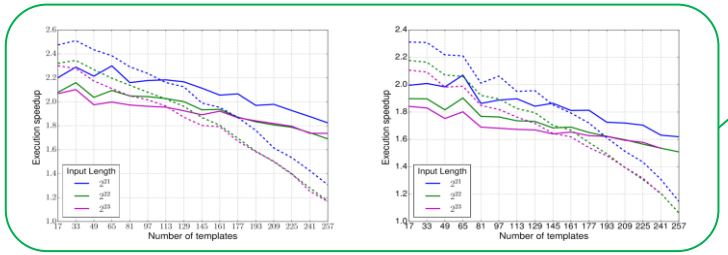




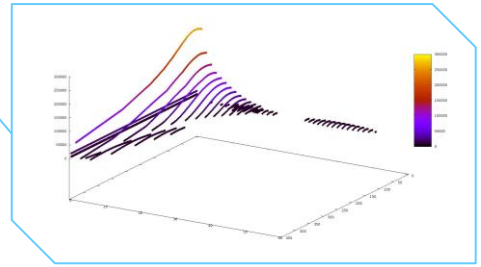


De-dispersion

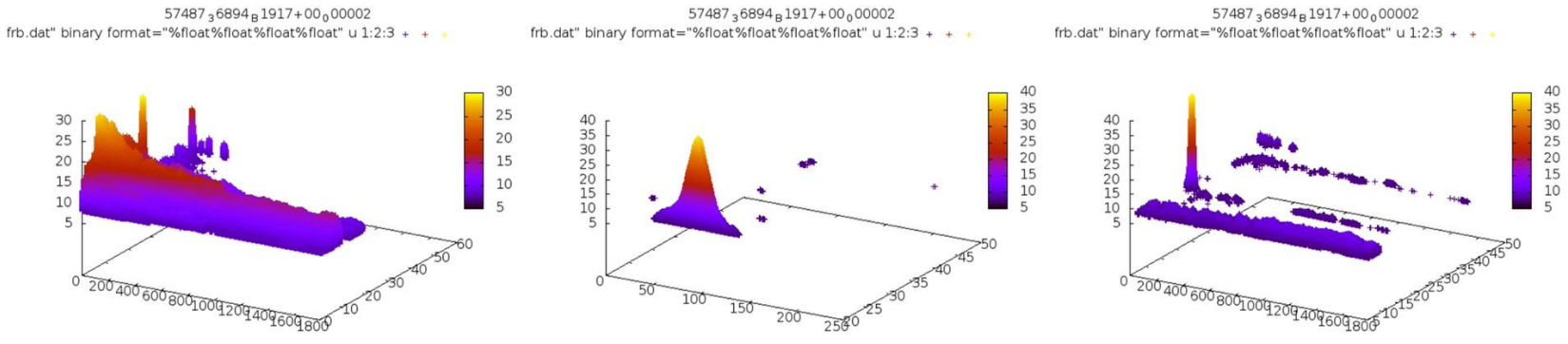
Fourier Domain Acceleration search



Periodicity Search

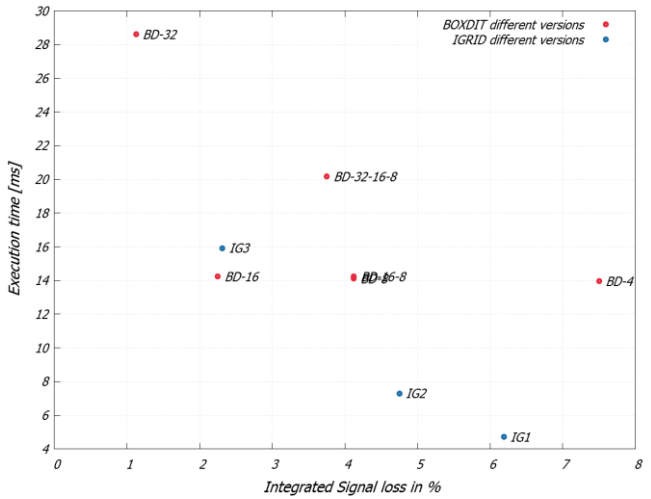


AstroAccelerate - Features



AstroAccelerate has the following features…

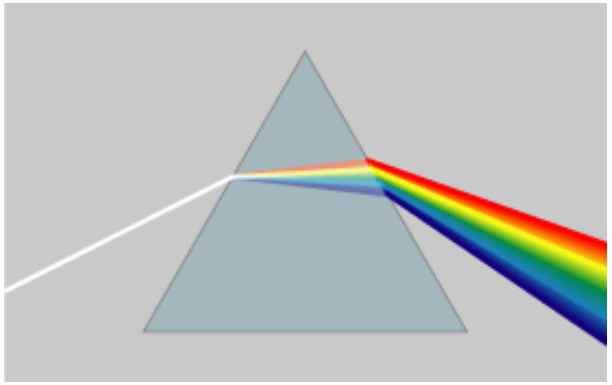
- Zero DM and basic RFI Mitigation
- DDTR
- Single Pulse Search
- Fourier Domain Acceleration Search
- Periodicity search with harmonic sum



Case study: Real-time de-dispersion for the SKA

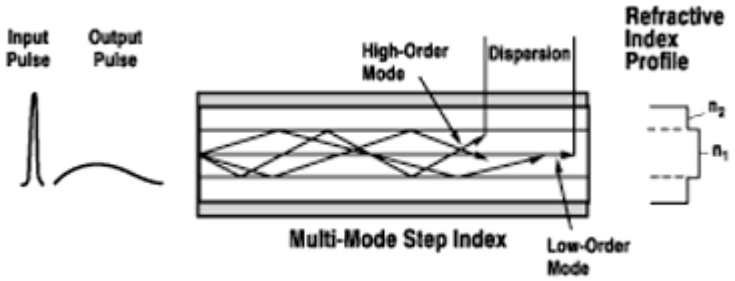
Mike Giles (Oxford)
Jan Novotný (Oxford)
Karel Adámek (Oxford)
Kate Clark (NVIDIA)
Tom Bradley (NVIDIA)
Tim Lanfear (NVIDIA)

What is dispersion?



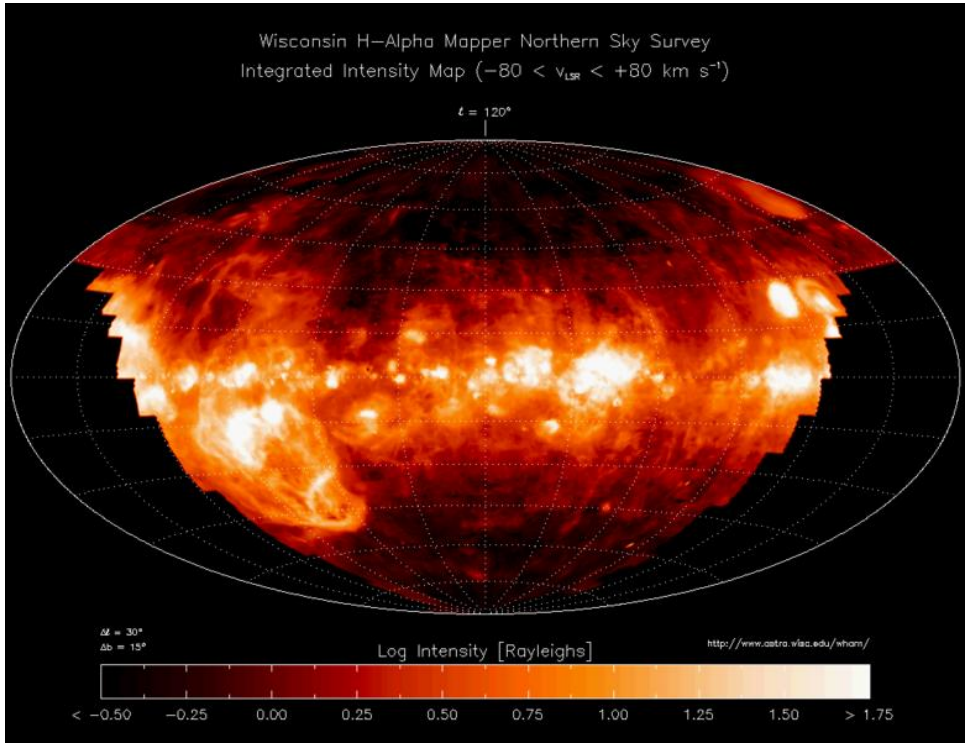
Chromatic dispersion is something we are all familiar with. A good example of this is when white light passes through a prism.

Group velocity dispersion occurs when pulse of light is spread in time due to its different frequency components travelling at different velocities. An example of this is when a pulse of light travels along an optical fibre.



Dispersion by the ISM

In our case, when thinking about a pulsar, the radio waves emitted from it can be dispersed…
The interstellar medium (ISM) is the matter that exists between stars in a galaxy.

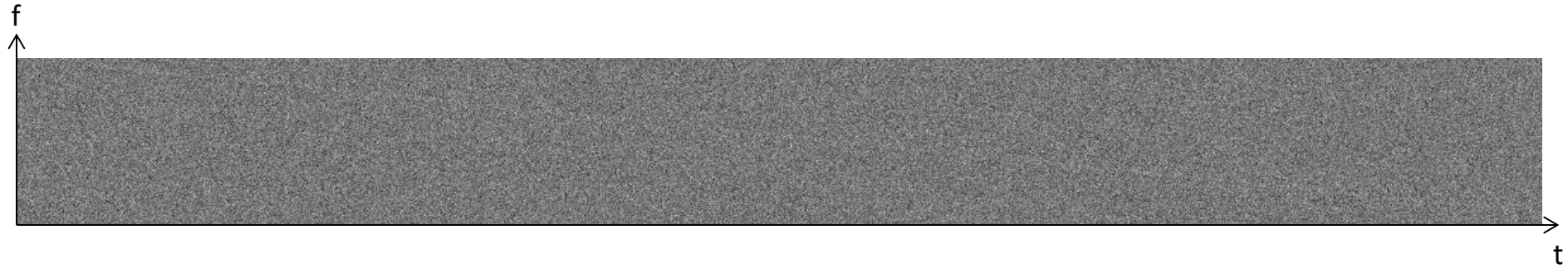


Haffner et al. 2003

In warm regions of the ISM (~8000K) electrons are free and so can interact with and affect radio waves that pass through it.

Experimental Data

When looking at a pulsar using a radio telescope, what do we see?



We “see” the above, this is our data.

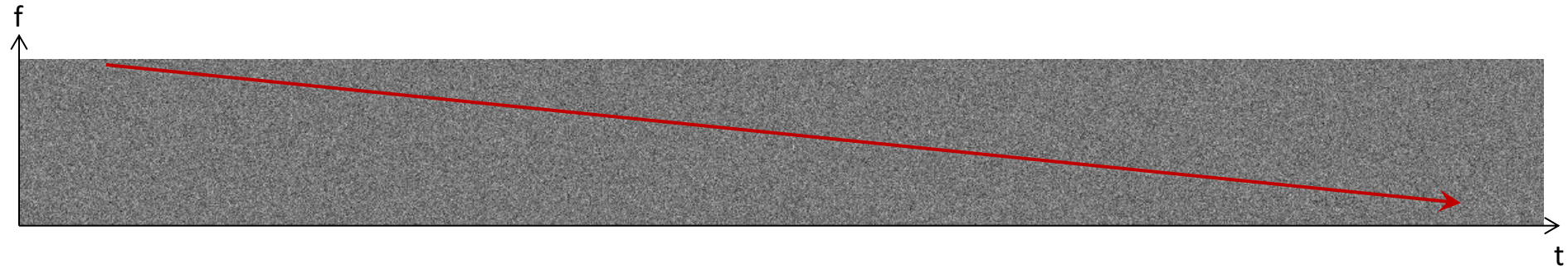
Each value on the y-axis represents a different value of frequency.

Each value on the x-axis represents a different value of time.

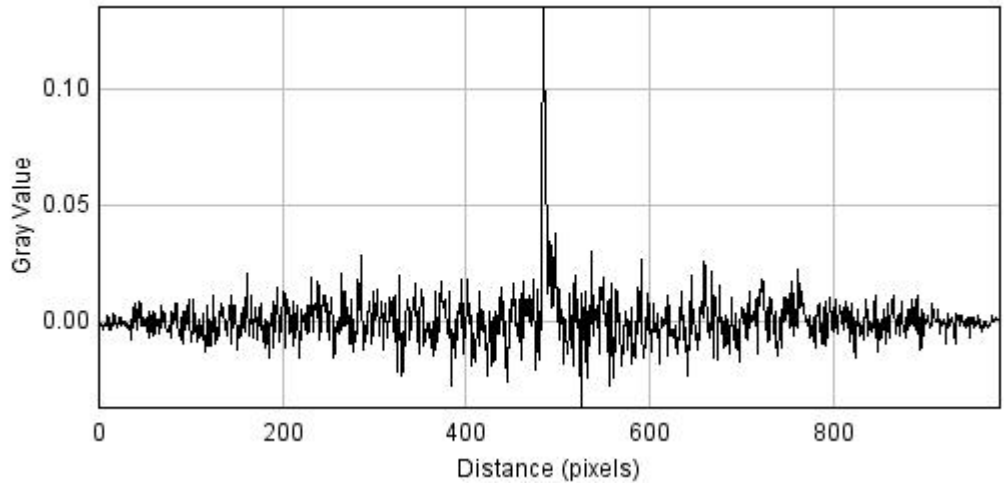
The “pixel” or $P(x, y) = P(t, f)$ value is a power measured by the telescope.

Most of the measured signals live in the noise of the apparatus.

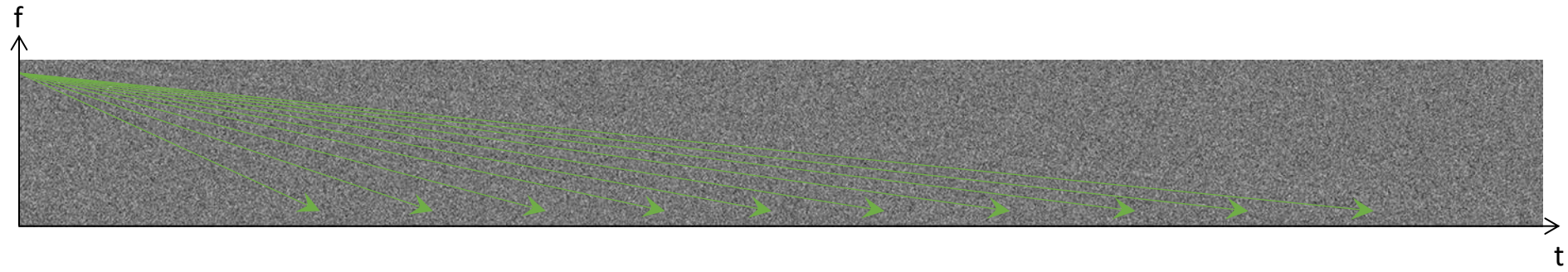
Experimental Data



So, if we sum each $P(t, f)$ that has a small amount of signal in it,
then we can lift the signal out of the noise so it can be seen (detected).



De-dispersion



In a **blind search** for a signal, we don't know when the signal will be emitted

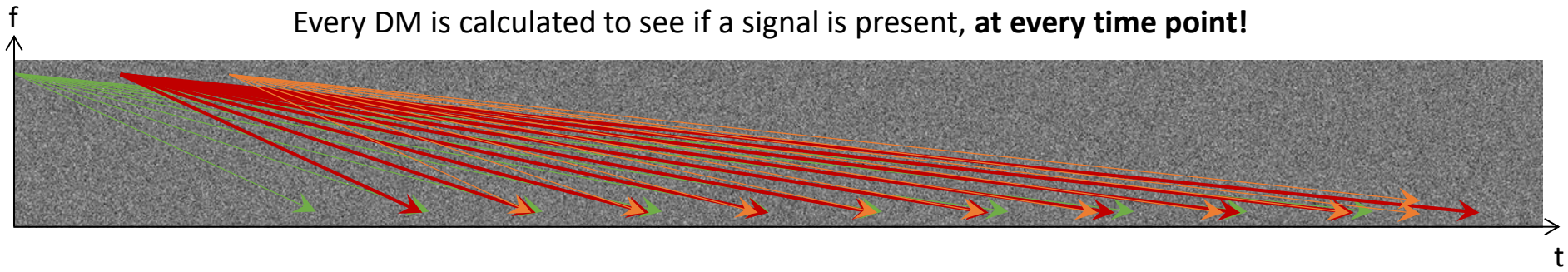
(it's location in time, from the point of view of our data)

Or, how far away the emitting pulsar is

(the distance relates to the steepness of the curve, *called the dispersion measure*, traced out by our signal)

To find out how far way the emitting object is, every dispersion measure is calculated to see if a signal is present.

De-dispersion



In a blind search for a signal many different dispersion measures are calculated.

This results in many data points in the (t,f) domain being used multiple times for different dispersion searches.

This allows for data reuse in a GPU algorithm.

All of this must happen in real-time i.e. the time taken to process all of our data must not exceed the time taken to collect it

The dispersion measure - DM

The time delay, $\Delta\tau$, between the detection of frequency f_{high} and f_{low} is given by:

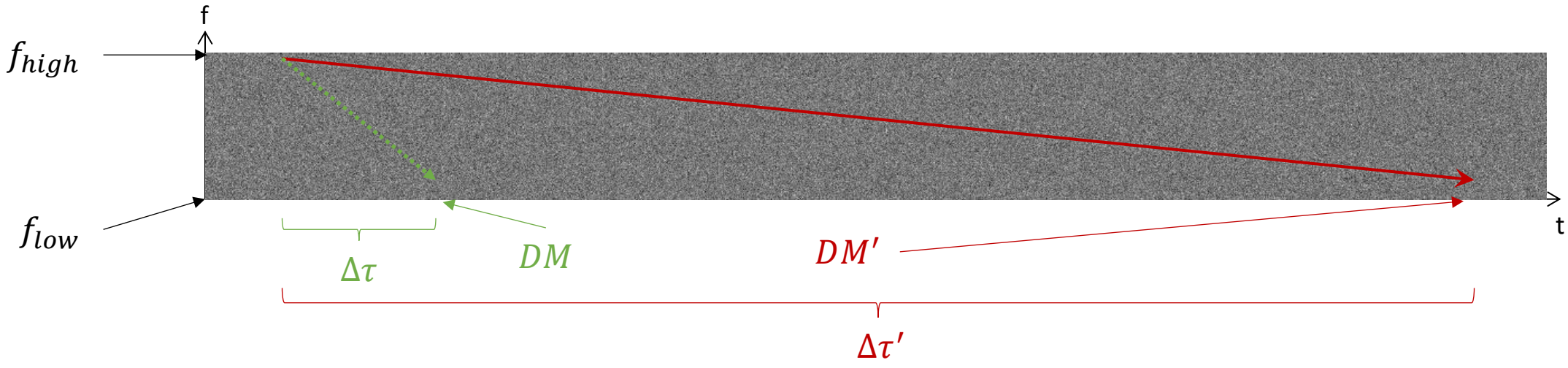
$$\Delta\tau = C_{DM} \times \mathrm{DM} \times \left(\frac{1}{f_{low}^2} - \frac{1}{f_{high}^2} \right)$$

Where C_{DM} is the dispersion constant. DM is the dispersion measure:

$$DM = \int_0^d n_e dl$$

This is the free electron column density between the radio source and observer.

In terms of our data



$$\Delta\tau = \mathrm{C}_{DM} \times \mathrm{DM} \times \left(\frac{1}{f_{low}^2} - \frac{1}{f_{high}^2}\right)$$

$$DM = \int_0^d n_e dl$$

So, what do we see above?

Our frequencies are set, they do not change, we choose those when starting our telescope.

C_{DM} this is a constant, the Universe has chosen that.

Hence our $\Delta\tau$ is only dependent on DM everything else is constant.

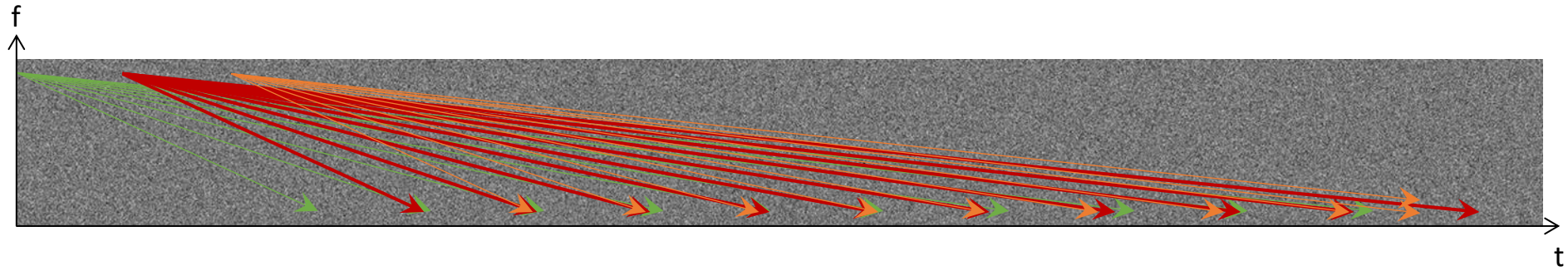
n_e think of this as the average number of electrons per meter along the line of sight between observer and pulsar, it's constant.

DM this can change, it is dependent on d the distance between observer and object, lower means steeper gradient of signal.

How do we turn this into a code?

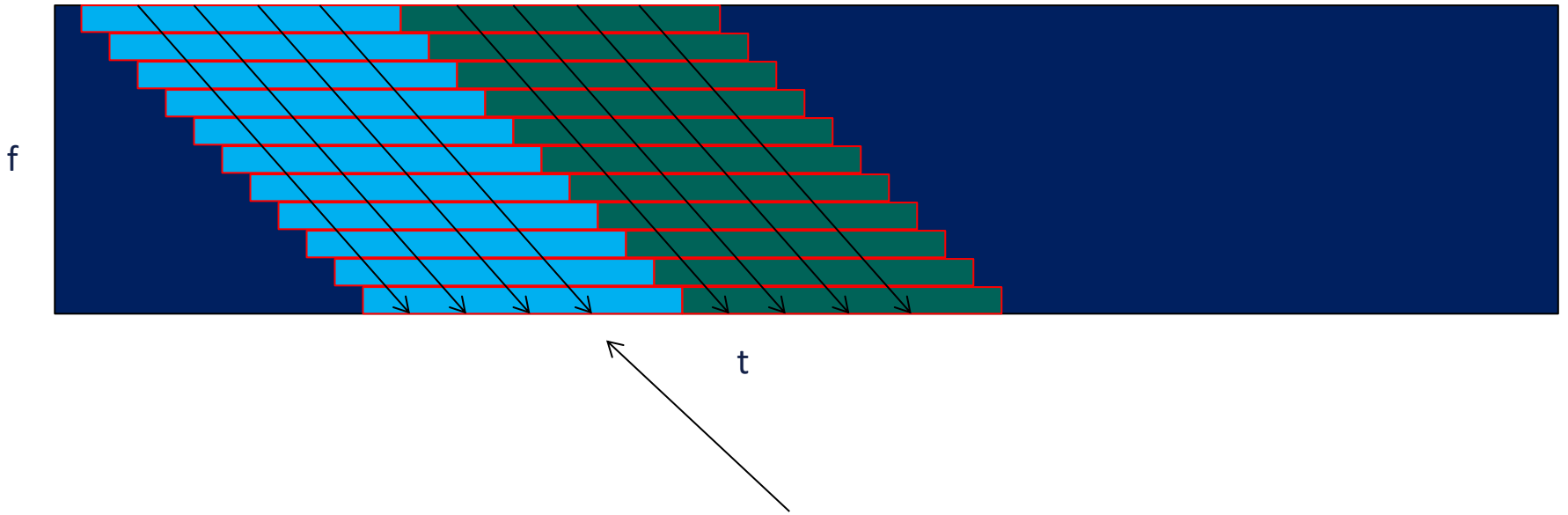
First, let's remind ourselves of what we need to do…

1. For all time values in our observation….
2. Pick a "distance" or DM to test (see if we have a signal).
3. For a given frequency, calculate a frequency dependent shift to pick the correct $P(t, f)$ input value to add to a running sum, for the fixed DM value we are investigating.



De-dispersion Transform

Each dispersion measure for a given frequency channel needs a shifted time value.



If I arrange my data so that time runs quickest (time contiguous) then for a constant DM value with varying time, each time value is contiguous in a cacheline.

A simple C - Code

```
// for each time sample
for(t = 0; t < (nsamp - maxshift); t++) {
    // calculate all dispersion trials
    for(dm_count = 0; dm_count < tdms; dm_count++) {
        // by summing over the specific time, frequency values that contribute to that DM
        for(c = 0; c < nchans; c++) {
            // calculate a "global" time dependent shift to our first cacheline of data
            int shift = ((c * nsamp) + t);

            // calculate a "local" shift within the cacheline of data
            float shift_temp = dm_count * (dm_step/tsamp);
            float dmshifts = 4148.741601 * ((1.0 / (fch1 + (foff * c)) / (fch1 + (foff * c))) - (1.0 / fch1 / fch1));

            // Caclulate the final shift using local and global shifts
            shift += floor(dmshifts * shift_temp);

            // Add the channel value "c" to the running sum for this dm_count
            output_buffer[(dm_count)*(nsamp-maxshift) + t] += input_buffer[shift];
        }
    }
}
```

Notes:

1. I've flattened the array, by this I mean that I have converted the 2D input to a contiguous 1D array.
2. nsamp is the number of time samples I have, tdms is the total number of tests (de-dispersions), nchans the number of frequency channels, tsamp the time sampling, fch1 the highest frequency and dm_step, the step size in DM.
3. My "shift_temp" value is just my discretised DM value.

Based on what we have taught you so far this week, is this a good implementation of my algorithm?

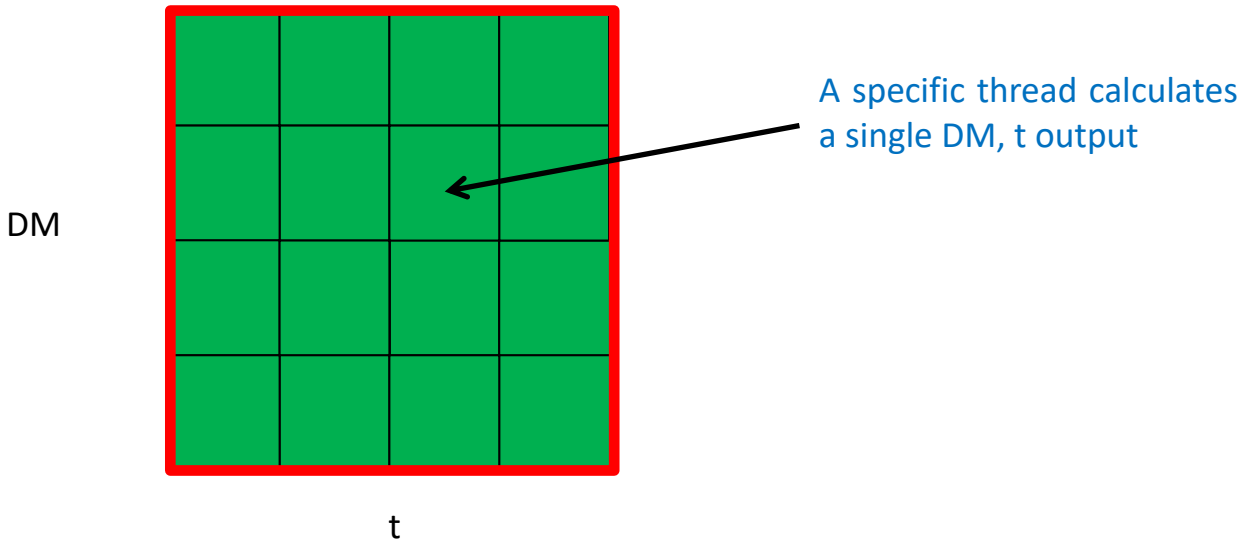
1. How do we turn this into a CUDA code?

Let's make some simple observations about our problem…

1. Our output "space" is a 2D array:
   - The x-axis is unchanged, it is time.
   - The y-axis is changed, now it is DM

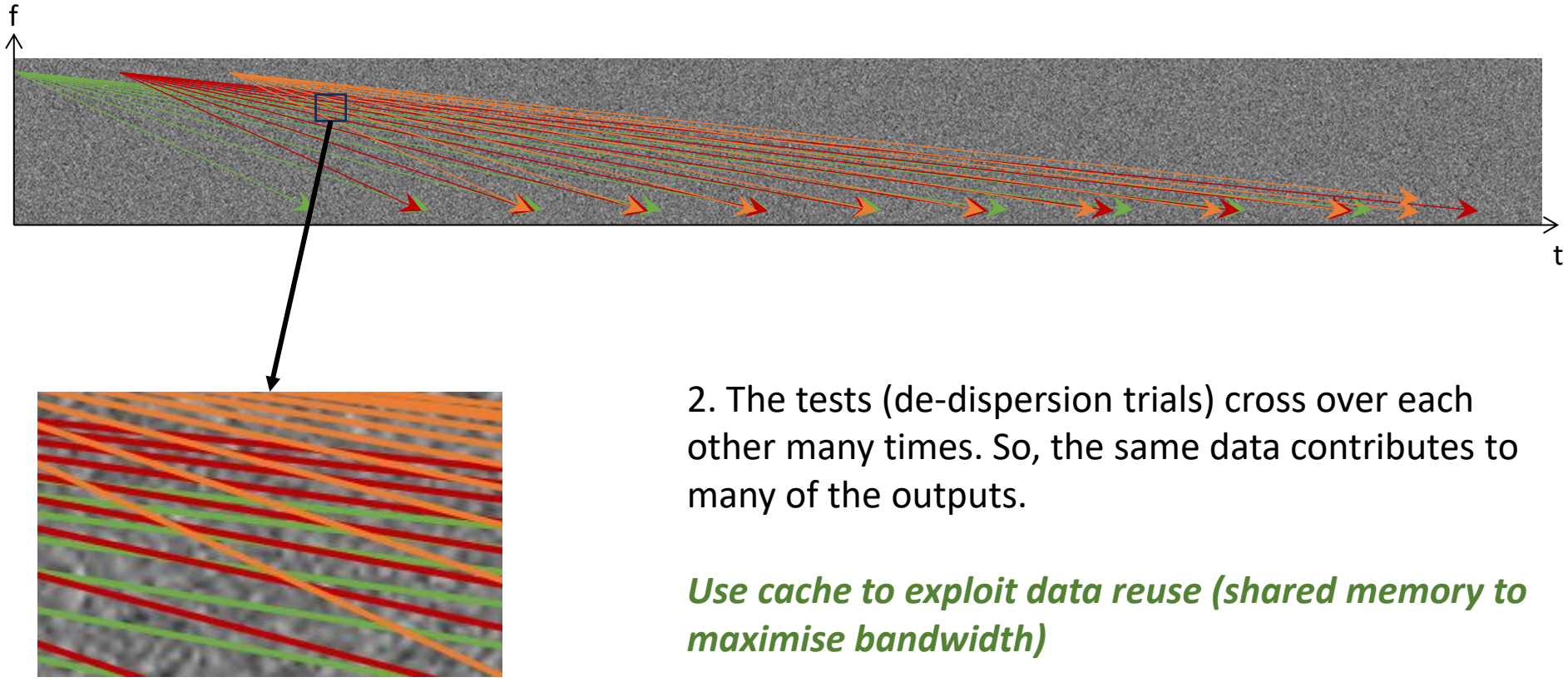
So, to map our grid of threads to our problem,

Let each thread represent one element of the output array.



2. How do we turn this into a CUDA code?

Looking back at our pictorial representation of our algorithm, what do we see?



2. The tests (de-dispersion trials) cross over each other many times. So, the same data contributes to many of the outputs.

Use cache to exploit data reuse (shared memory to maximise bandwidth)

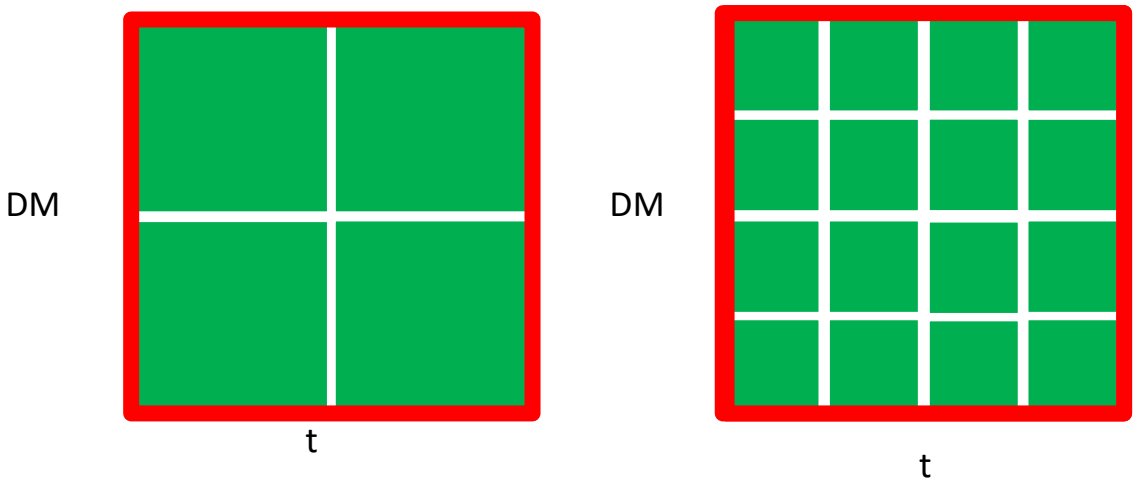
3. How do we turn this into a CUDA code?

Our last slide showed us that different de-dispersion trials (having different time values and different DM values) cross over each other, providing opportunity for data reuse.

For us to exploit this we said we can use cache or shared memory.

3. How do we maximise data reuse?

We can change our thread block size to maximise our data reuse. This controls the number of de-dispersion trials and number of time steps each thread block computes, and so what associated data is in cache (or shared memory….)



Optimising the parameterisation

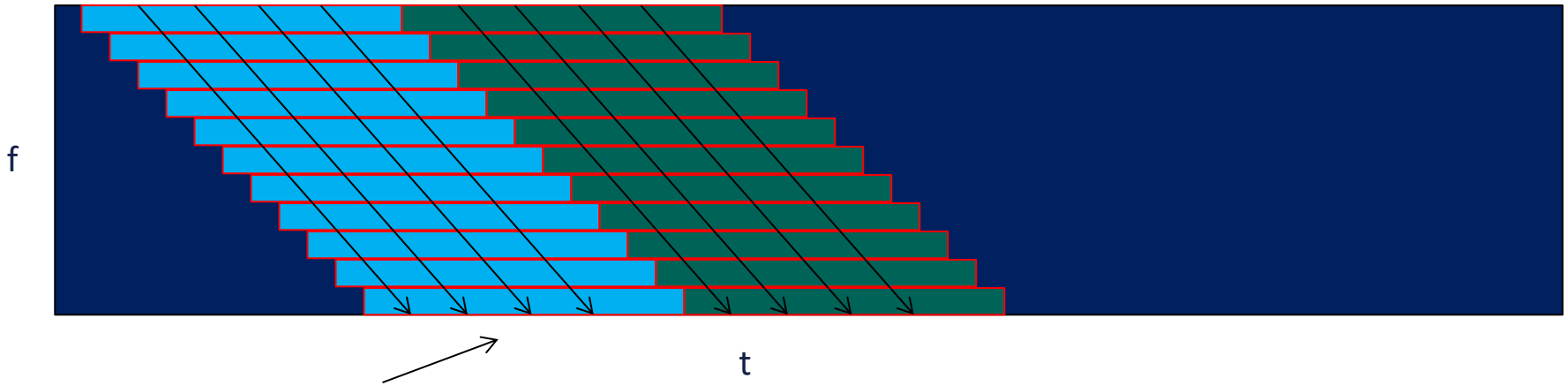
A simple CUDA - Code

```
__global__ void cache_dedisperse_kernel(int bin, unsigned short *d_input, float *d_output, float mstartdm, float mdmstep) {
  size_t   shift;
  float local_kernel;

  int t  = blockIdx.x * SDIVINT  + threadIdx.x;
  // Initialise the time accumulators
  local_kernel = 0.0f;

  float shift_temp = mstartdm + ((blockIdx.y * SDIVINDM + threadIdx.y) * mdmstep);

  // Loop over the frequency channels.
  for(int c = 0; c < i_nchans; c++) {


    // Calculate the initial shift for this given frequency
    // channel (c) at the current despersion measure (dm)
    // ** dm is constant for this thread!!**

    shift = ((size_t)(c)*(size_t)(i_nsamp) + (size_t)(t)) + (size_t)(__float2int_rz (dm_shifts[c] * shift_temp));

    local_kernel += (float)__ldg(&d_input[shift]);
  }

  // Write the accumulators to the output array.
  shift = ( ( (size_t)( blockIdx.y) * (size_t)(SDIVINDM ) + (size_t)threadIdx.y ) * ( (size_t)i_t_processed_s ) ) + (size_t)t;

  d_output[shift] = (local_kernel / i_nchans / bin);

}
```

4. How do we turn this into an optimised CUDA code?

4. How do we exploit registers to increase speed?

Each dispersion measure for a given frequency channel needs a shifted time value.



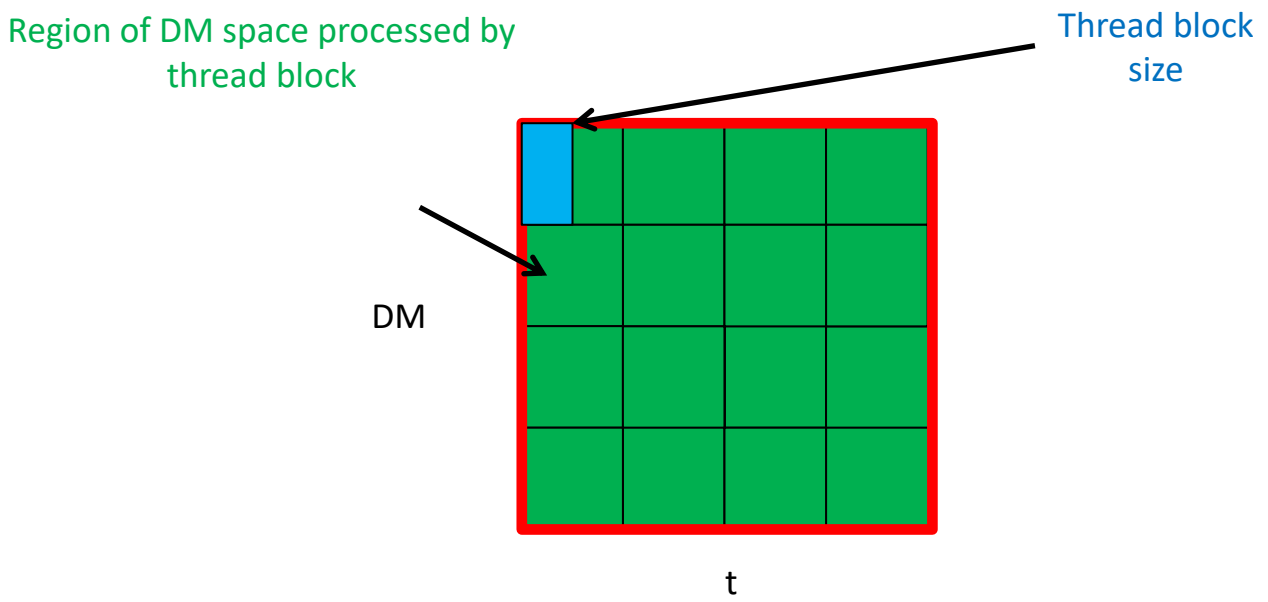
Constant DM's with varying time.

In practice a thread will process multiple time samples, and a thread block will also process neighboring DM trials to increase data reuse.

4. How do we turn this into an optimised CUDA code?

Each thread processes a tunable number of time samples; each **de-dispersion trial associated with one time sample is stored in a GPU register**.

Along with this **the number of time samples per thread block and the number of de-dispersion trials (which is where data reuse comes from) are tuned.**



https://github.com/AstroAccelerateOrg/astro-accelerate/blob/master/lib/device_dedispersion_kernel.cu

A simple++ CUDA - Code

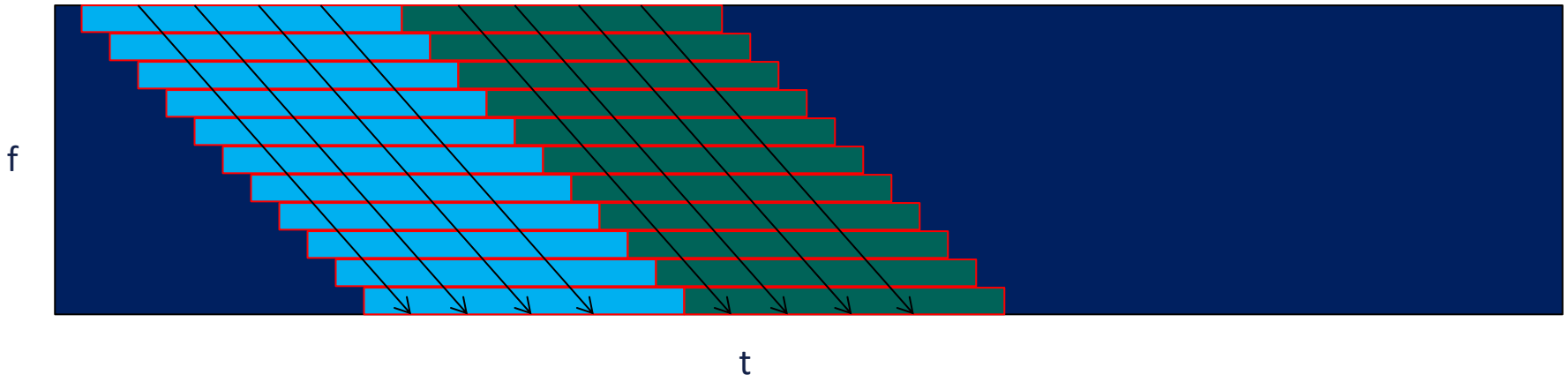
```
__global__ void cache_dedisperse_loop(float *outbuff, float *buff, float mstartdm, float mdmstep)
{

    // NOTE: inshift AND outshift are set to 0 (zero) in the kernel call and so is
    // removed from this kernel.

    int   shift;
    float local_kernel_t[NUMREG];

    int t  = blockIdx.x * NUMREG * DIVINT  + threadIdx.x;

    // Initialise the time accumulators
    for(int i = 0; i < NUMREG; i++) local_kernel_t[i] = 0.0f;

    float shift_temp = mstartdm + ((blockIdx.y * DIVINDM + threadIdx.y) * mdmstep);

    // Loop over the frequency channels.
        for(int c = 0; c < i_nchans; c++) {


        // Calculate the initial shift for this given frequency
        // channel (c) at the current despersion measure (dm)
        // ** dm is constant for this thread!!**
        shift = (c * (i_nsamp) + t) + __float2int_rz (dm_shifts[c] * shift_temp);

        #pragma unroll
        for(int i = 0; i < NUMREG; i++) {
            local_kernel_t[i] += buff[shift + (i * DIVINT) ];
        }
    }

    // Write the accumulators to the output array.
    #pragma unroll
    for(int i = 0; i < NUMREG; i++) {
        outbuff[((blockIdx.y * DIVINDM) + threadIdx.y)* (i_nsamp-i_maxshift) + (i * DIVINT) + (NUMREG * DIVINT * blockIdx.x) + threadIdx.x] = local_kernel_t[i];
    }

}
```

This new kernel is still cache based; However, we process NUMREG time trials per thread (note the stride by DIVINT)

5. How do we turn this into an optimised CUDA code?

5. How do we exploit fast shared memory?

Each dispersion measure for a given frequency channel needs a shifted time value.



Incrementing all of the registers at every frequency step ensures a high data reuse of the stored frequency time data in the L1 cache or shared memory.

A slightly more complicated CUDA - Code

```
__global__ void shared_contiguous_loop(float *outbuff, float *buff, float mstartdm, float mdmstep)
{

    int   i, c, shift;
    float local_kernel_t[NUMREG];

    // Initialise the time accumulators
    for(i = 0; i < NUMREG; i++) local_kernel_t[i] = 0.0f;

    int shift_one = (mstartdm +((blockIdx.y*DIVINDM + threadIdx.y)*mdmstep));
    int shift_two = (mstartdm + (blockIdx.y*DIVINDM*mdmstep));
    int idx = (threadIdx.x + (threadIdx.y * DIVINT));
        for(c = 0; c < i_nchans; c++) {

        f_line[idx] = buff[((c*i_nsamp) + (blockIdx.x*NUMREG*DIVINT + idx)) + __float2int_rz(dm_shifts[c]*shift_one)];
        __syncthreads();

        shift = __float2int_rz(dm_shifts[c]*shift_one) - __float2int_rz(dm_shifts[c]*shift_two) + threadIdx.x*NUMREG;

        #pragma unroll
        for(i = 0; i < NUMREG; i++) {
            local_kernel_t[i] += f_line[(shift + i)];
        }
    }

    // Write the accumulators to the output array.
    shift = ((blockIdx.y*DIVINDM) + threadIdx.y)*(i_nsamp-i_maxshift) + (NUMREG*DIVINT*blockIdx.x) + threadIdx.x*NUMREG;

    #pragma unroll
    for(i = 0; i < NUMREG; i++) {
        outbuff[shift + i] = local_kernel_t[i];
    }
}
```

Each thread (having local index idx) loads one element form the input array into a staging area in shared memory called f_line[] this is then used to update the local accumulators local_kernel_t[]

6. Extreme optimisation?

We can exploit the fact that **one frequency-time sample of SKA telescope data will be 8 bits**.

So currently our code moves 32 bits (our input is currently float but actually has 8 bits of information in it.

Our native “word size” on the GPU is 32 bits, it is optimised to do 32-bit computation.

6. Given our data is 8 bits, but we are performing 32-bit operations, can we perform more than one operation per 32-bit computation?

Recorded telescope data (t_n = 8 bits) is stored as a uchar array

char[] = [t_0,t_1,t_2,t_3,t_4,t_5,t_6 …]

6. Extreme optimisation?

We exploit the fact that **one frequency-time sample of SKA data will be 8 bits**.

We **pack the data in such a way so that we can perform two de-dispersion trials per integer operation**.

We convert the **unsigned char to an unsigned short and pack as ushort2**, we **mask this as an int and add ints**.

Once a single trial nears the maximum allowable value for a ushort we store the value in a floating point accumulator. This has the effect of increasing the speed of the code and also it's precision.

Recorded telescope data (t_n = 8 bits) is stored in global as a uchar array

char[] = [t_0,t_1,t_2,t_3,t_4,t_5,t_6 …]

This is converted to ushort when loaded though the texture pipe (doubling the size of the array stored because it is now interleaved with 8 bits of zeros

ushort[] = [0 t_0, 0 t_1, 0 t_2, 0 t_3, 0 t_4, 0 t_5, 0 t_6, …]

Masking this with an int allows us to add two samples per one instruction issued.

6. Extreme optimisation?

In reality we must odd/even interleave the data to ensure correct byte alignment within shared memory banks (4 bytes wide).

ushort2[] = [0 t_0,0 t_1][0 t_1,0 t_2][0 t_2,0 t_3][0 t_3,0 t_4]…

For thread with an even shift (let's say 2)…

(t_0,t_1) (t_1,t_2) (t_2,t_3) (t_3,t_4) (t_4,t_5) (t_5,t_6)

$t_i = t_2$

$t_{i+1} = t_3$

For thread with an odd shift (let's say 3)…

(t_0,t_1) (t_1,t_2) (t_2,t_3) (t_3,t_4) (t_4,t_5) (t_5,t_6)

$t_i = t_3$

$t_{i+1} = t_4$

Now each thread computes the correct time values (two per word) and at double data rate

A complicated CUDA - Code

```
for (c = 0; c < i_nchans; c += UNROLLS)
  {

    __syncthreads();

    for (j = 0; j < UNROLLS; j++)
      {
        temp_f = ( __ldg(( d_input + ( __float2int_rz(dm_shifts[c + j] * shift_two) ) )  + ( nsamp_counter + ( j * i_nsamp ) )) );

        f_line[j][idx + 1].x = temp_f;
        f_line[j][idx    ].y = temp_f;

        shift[j] = __float2int_rz(shift_one * dm_shifts[c + j] + findex);
      }

    nsamp_counter = ( nsamp_counter + ( UNROLLS * i_nsamp ) );

    __syncthreads();

    for (i = 0; i < SNUMREG; i++)
      {
        local = 0;
        unroll = ( i * 2 * SDIVINT );
        for (j = 0; j < UNROLLS; j++)
          {
            stage = *(int*) &f_line[j][( shift[j] + unroll )];
            local += stage;
          }
        local_kernel_one[i] += (local & 0x0000FFFF);
        local_kernel_two[i] += (local & 0xFFFF0000) >> 16;
      }
  }
```

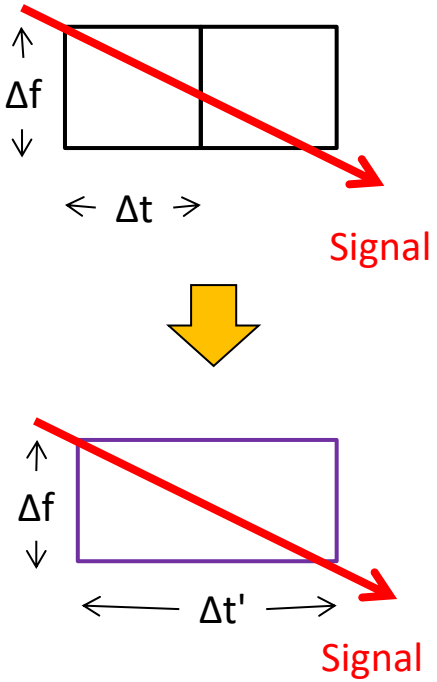
elerate/blob/master/lib/device_dedispersion_kernel.cu

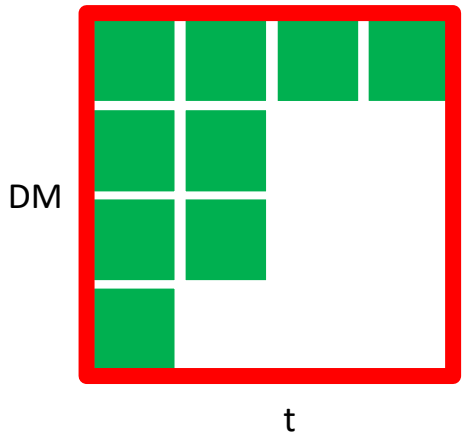
Exploiting the physics

One issue with using a shared memory based algorithm is that for high DM trials (those that represent distant objects, forming long broad curves in our input frequency-time data) **we need to store increasing lengths of constant frequency varying time data in shared memory.**

This ultimately limits the highest DM trial that can be searched at full time resolution.

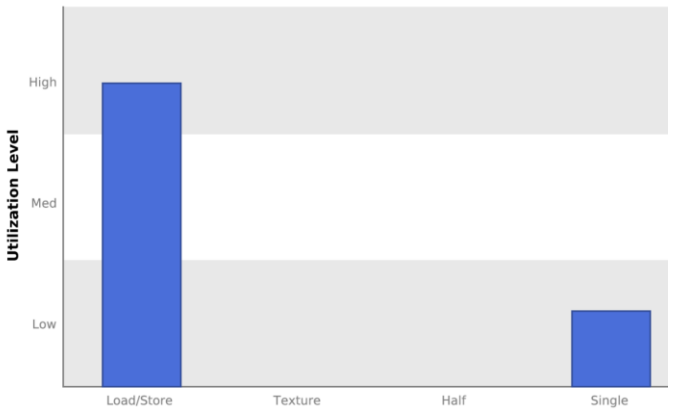
To overcome this, we've added a time binning (scrunching) kernel that decimates data in time. This has the effect of decreasing time resolution and allows us to search to arbitrary high DM trials.

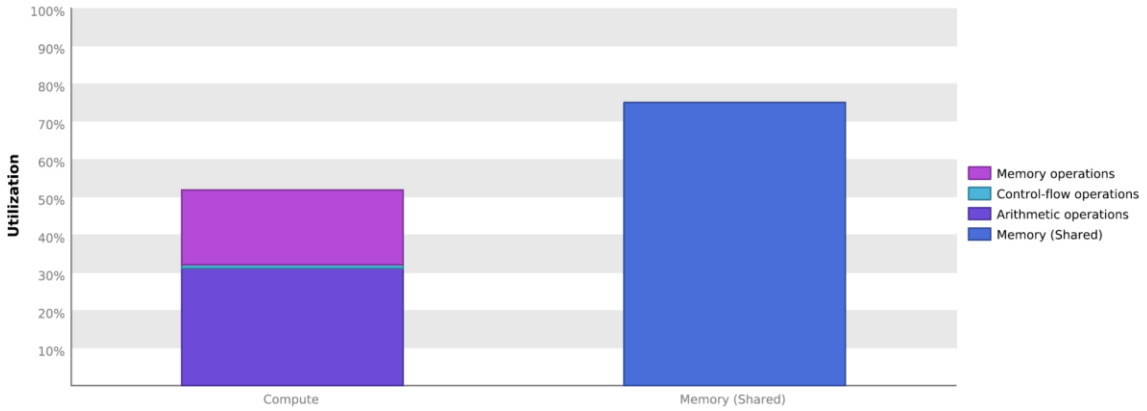




Has the added advantage of reducing the amount threads that are needed to process a region of (DM,t) space, speeding up the code.

De-dispersion Transform - results





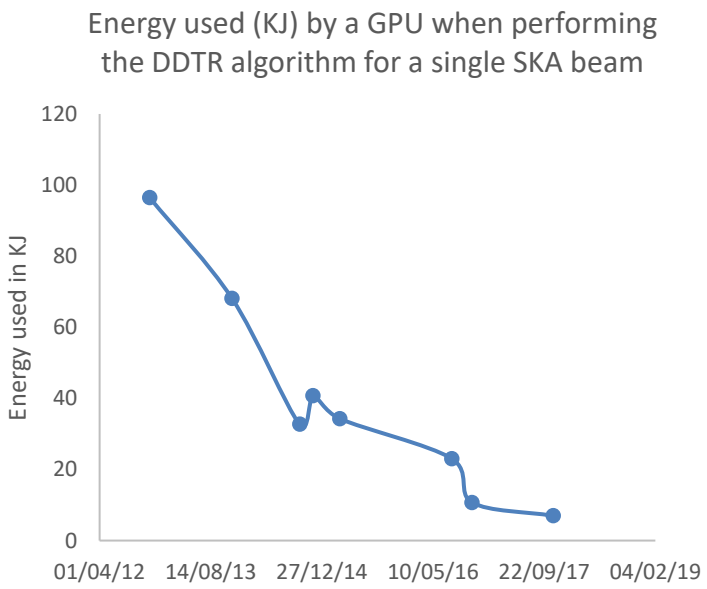
Shared Memory

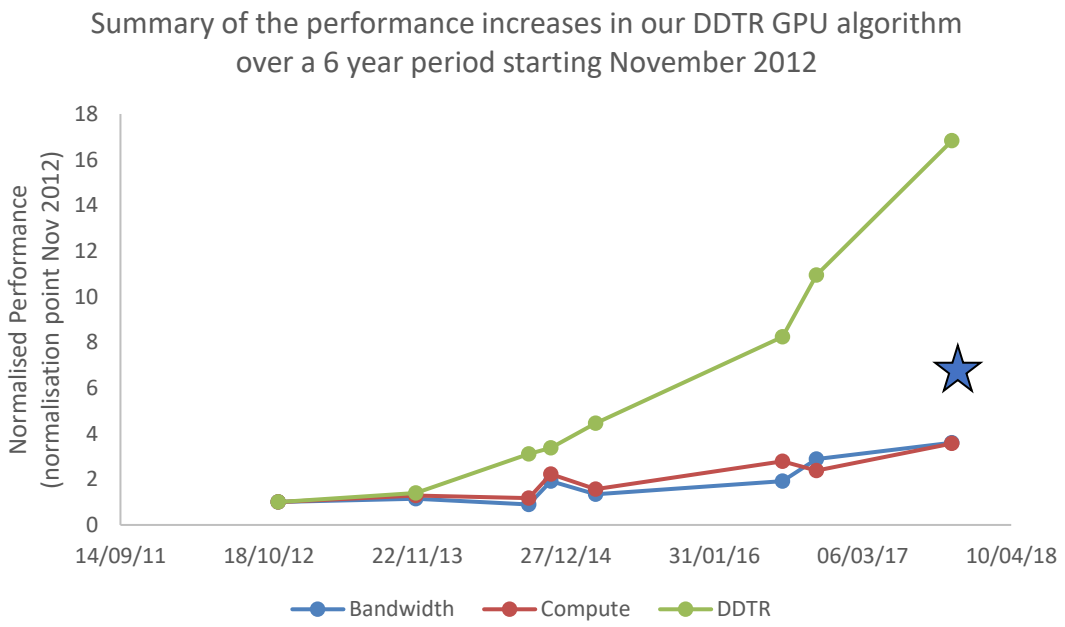
Shared Loads	43769001212	7,976.627 GB/s	
Shared Stores	5655113738	1,030.609 GB/s	
Shared Total	49424114950	9,007.236 GB/s	Idle Low Medium High Max

Results showing the shared memory utilisation, which is this codes limiting factor. **We achieve 75% of peak throughput**, limited by load/store.

The total shared memory bandwidth throughput achieved on a **TITAN V is 9 TB/s**.

De-dispersion Transform - results





These two plots demonstrate how we have reduced power consumption and increased performance for the DDTR algorithm over a six year period.

The bule star indicates the performance of our initial (optimised) code running on current hardware. Demonstrating how invested effort algorithm optimisation over a long period can deliver significant gains.

De-dispersion Transform – cost / benefit analysis

But is it worth the effort?

Estimated runtime for DDTR in the PSS pipeline (conservative 25%)
Estimate of speed increase compared to initial code ~17x

➡ Total PSS pipeline acceleration ~ 4x

So to deliver the science in the same wall clock time you'd need 4x the GPU capacity.

Even if you're prepared to wait 4x longer… Energy efficiency has increased by 14x
Very rough estimate of PSS OpEx saving ~ £1M
Estimate of total effort ~ 1.0FTE for four years ~ £250K (FEC)

Hence a £750K saving in OpEx costs alone (this is a conservative estimate).

(You can't just go out and buy this at a later date. Domain expertise in both radio astronomy data processing and many-core acceleration are needed)

Conclusions – Comparisons of GPUs

Technology	Kepler (K40)	Kepler (K80)	Kepler (780Ti)	Maxwell (980)	Maxwell (Titan X)	Pascal (Titan XP)	Pascal P100	Volta V100	Volta Titan V
Fraction of real-time	1.035	2.5	2.88	2.3	3.3	6.1	8.1	12.5	10.9
Watts per beam (Average)	127W	76 W	~70W	~61W	~64W	~43W	~24W	13W	10W
Cost per beam (capital, accelerator only)	£3K?	£4K?	£250	£200	£240	~£200	~£420	~£530	~£270
Cost per beam (2 year survey, GPU only, based on 1KWh costing £0.2)	~£430	~£265	~£245	~£213	~£224	~£151	~£84	~£45	~£35

Improvement between generations comes from a combination of advances in both the hardware and algorithm

A final word on measurement

Often nvidia-smi is used to establish values of physical parameters of the GPU whilst it is executing code.

If you have workloads with short duration kernels this might not be the best thing to do…

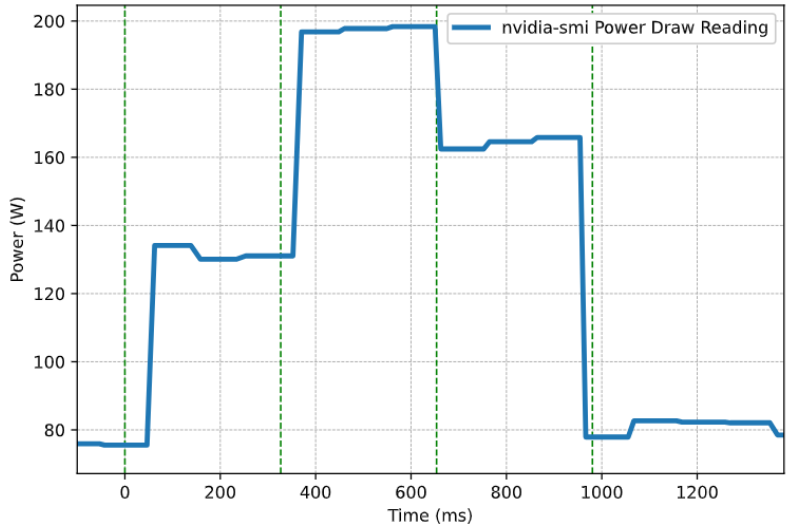


Figure 1: nvidia-smi can report drastically different power draw for the same CUDA Kernel, ranging from 80W to 200W. This is because nvidia-smi does not fully capture the power information on some GPUs. The figure shows a CUDA program that runs for 325ms on an A100 GPU. The kernel is executed 4 times, and the green dotted line indicates the beginning of each iteration.

Jim Yang et. al. (to appear in SC24 proceedings) https://arxiv.org/abs/2312.02741

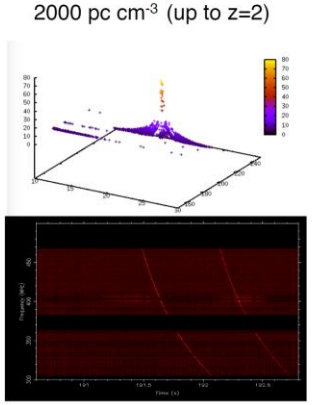
Impact

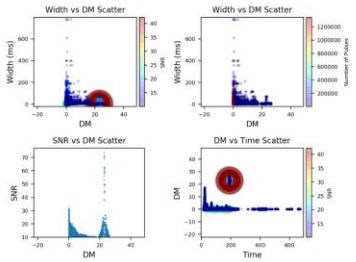
Giant Metrewave Radio Telescope

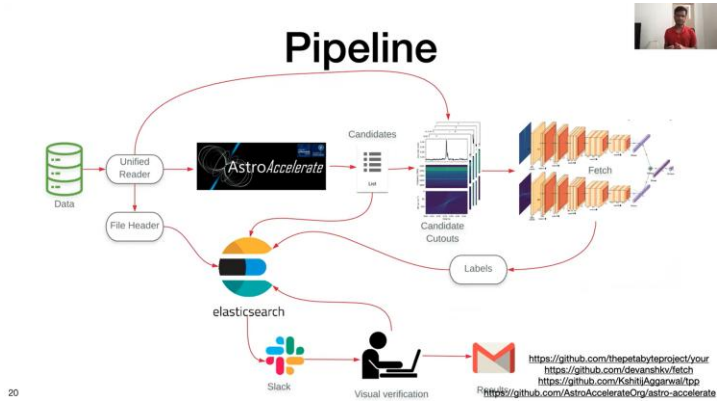
[Presentation]

Real-time FRB search pipeline for GHRSS

- A GPU-based FRB detection pipeline for the GMRT (using AstroAccelerate credit: Wes et al.) → 1st RRAT from GMRT discovered!!
- GMRT beams can be processed in (5x faster than) real-time for DM up to 2000 pc cm^{-3} (up to z=2)

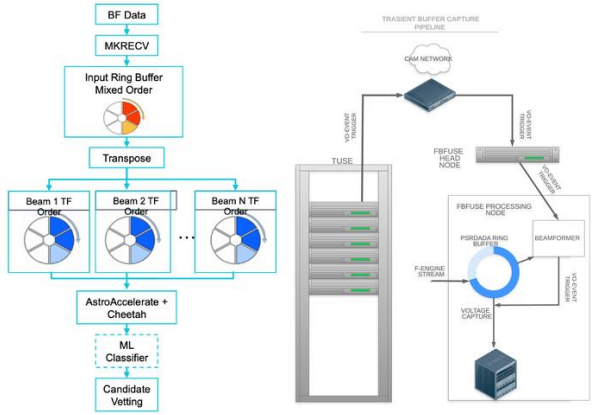






The Petabyte FRB Search Project

[Presentation]



MeerKAT (MeerTRAP)

[paper] ([arxiv])

Acknowledgments and Collaborators

University of Oxford

Mike Giles (Maths)

Aris Karastergiou (Physics)

Chris Williams (Physics)

Steve Roberts (Engineering)

Sofia Dimoudi (OeRC)

Nassim Ouannoughi (OeRC)

Karel Adamek (OeRC)

Jayanth Chennamangalam (Physics)

Griffin Foster (Physics)

University of Manchester

Ben Stappers

Mike Keith

Prabu Thiagaraj

Jayanta Roy

Mitch Mickaliger

NRAO/UVA

Scott Ransom

University of Bristol

Dan Curran (Electrical Engineering)

Simon McIntosh Smith (Electrical Engineering)

ASTRON

Cees Bassa

Jason Hessels

University of Opava

Jan Novotny

NVIDIA

Kate Clark

Tim Lanfear

Tom Bradley

Max Plank

Ewan Barr

AstroAccelerate

http://www.oerc.ox.ac.uk/projects/astroaccelerate