Using Rust in Radio Astronomy

Christopher Jordan

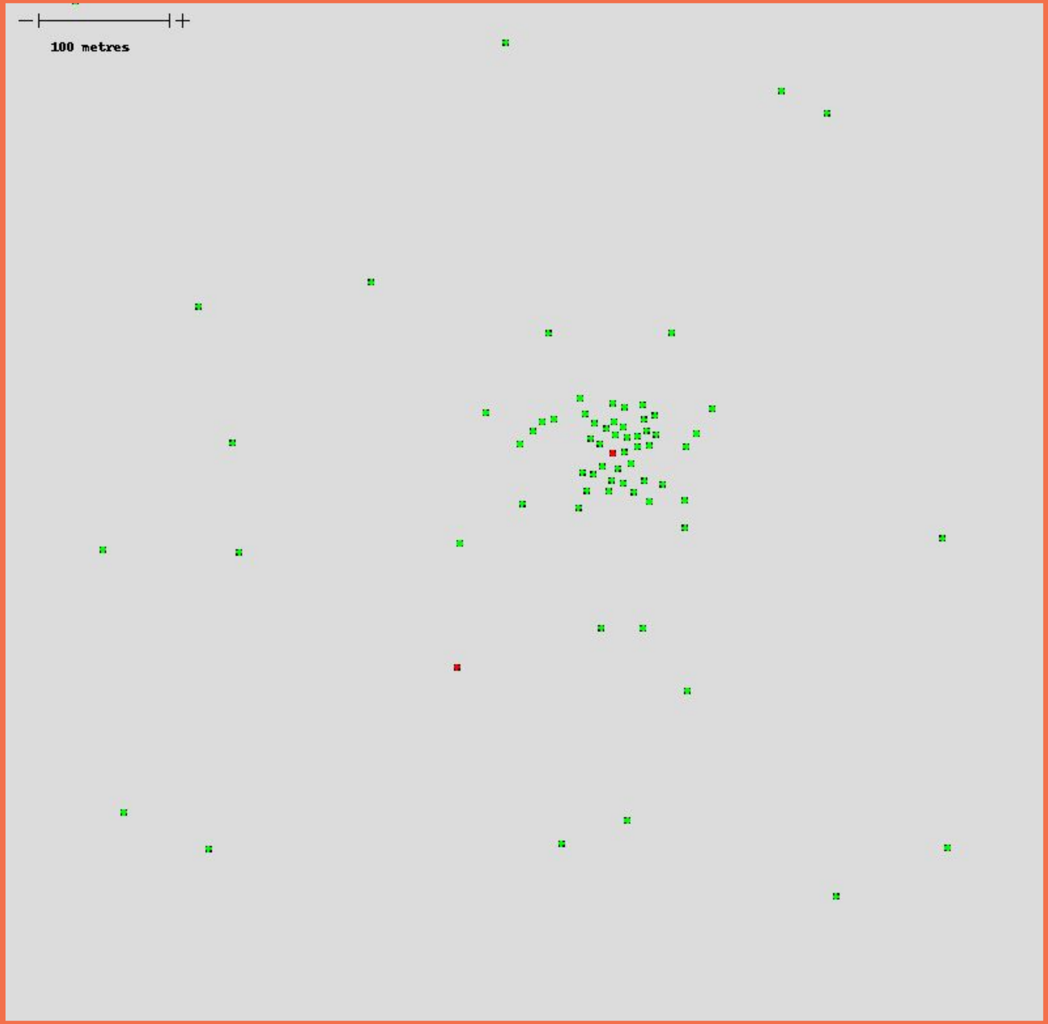
Curtin University, Australia

ASTRO 3D

ARC CENTRE OF EXCELLENCE FOR ALL SKY ASTROPHYSICS IN 3D

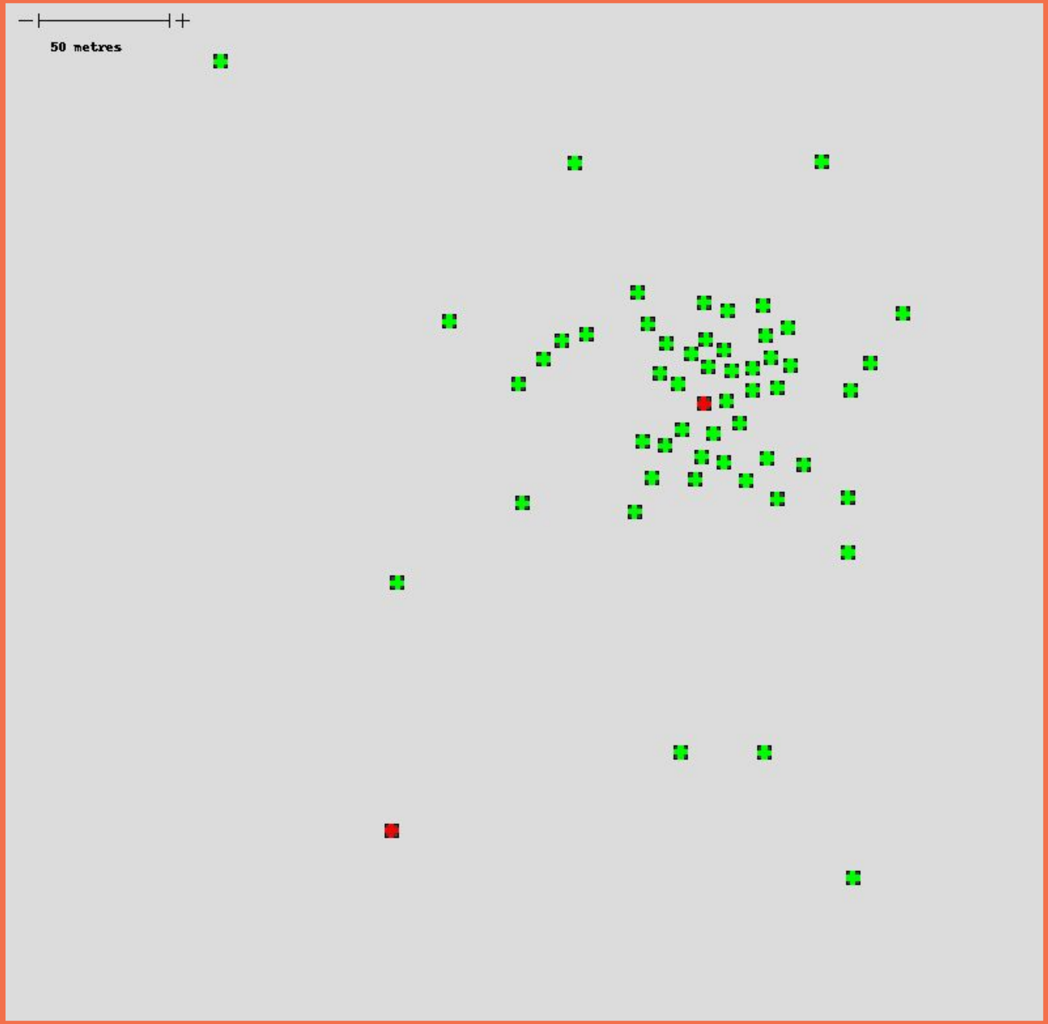
UNLOCKING THE UNIVERSE, INSPIRING THE FUTURE

The MWA


100 metres

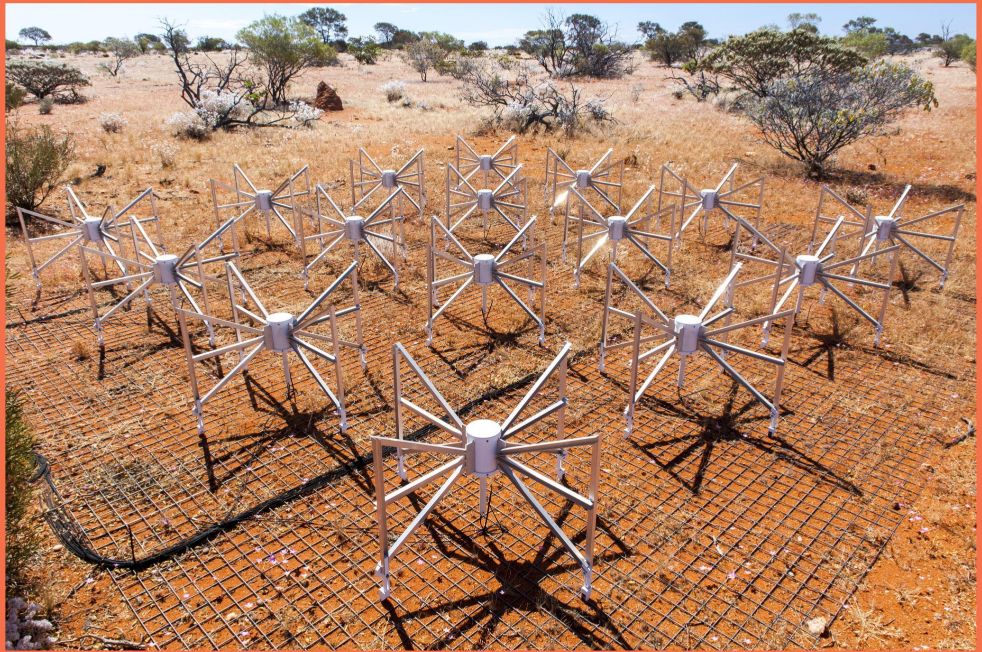

Drysdale River
Kununurra
Dampier Peninsula
Gibb
Warmun
Derby
Wunaamin Miliwundi Ranges
Ord River
Purnul
Broome
St George Ranges
Fitzroy Crossing
Halls Creek
Mueller Ranges
Eighty Mile Beach
Sturt Creek
Tanami
Port Hedland
Karratha
Marble Bar
Telfer
Onslow
Exmouth
Cape Range National Park
Mount Sheila
Karijini
Tom Price
Karlamilyi National Park
Gibson Desert North
Coral Bay
Paraburdoo
Newman
Patjarr
Carnarvon
Kumarina
Peak Hill
Little Sandy Desert
Gibson Desert Nature Reserve
Kaltuka
Warakurna
Denham
Mantamaru
Tjirrkarli
Warburton
Lake Carnegie
Murchison Widefield Array
Wiluna
WESTERN AUSTRALIA
Gibson Desert South
Beadell
Laverton
Geraldton
Leonora
Plumridge Lakes
Great Victoria Desert Nature Reserve
Kalgoorlie
Forrest
Eucla
Perth
Madura
Mundrabilla
Norseman
Rockingham
Esperance
Margaret River
Albany
Google

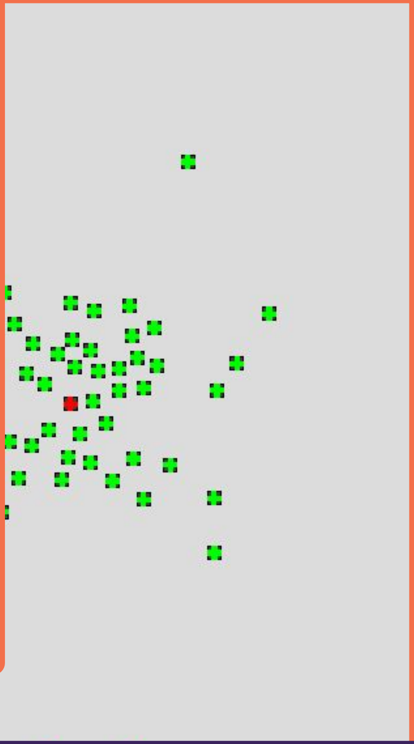
The MWA

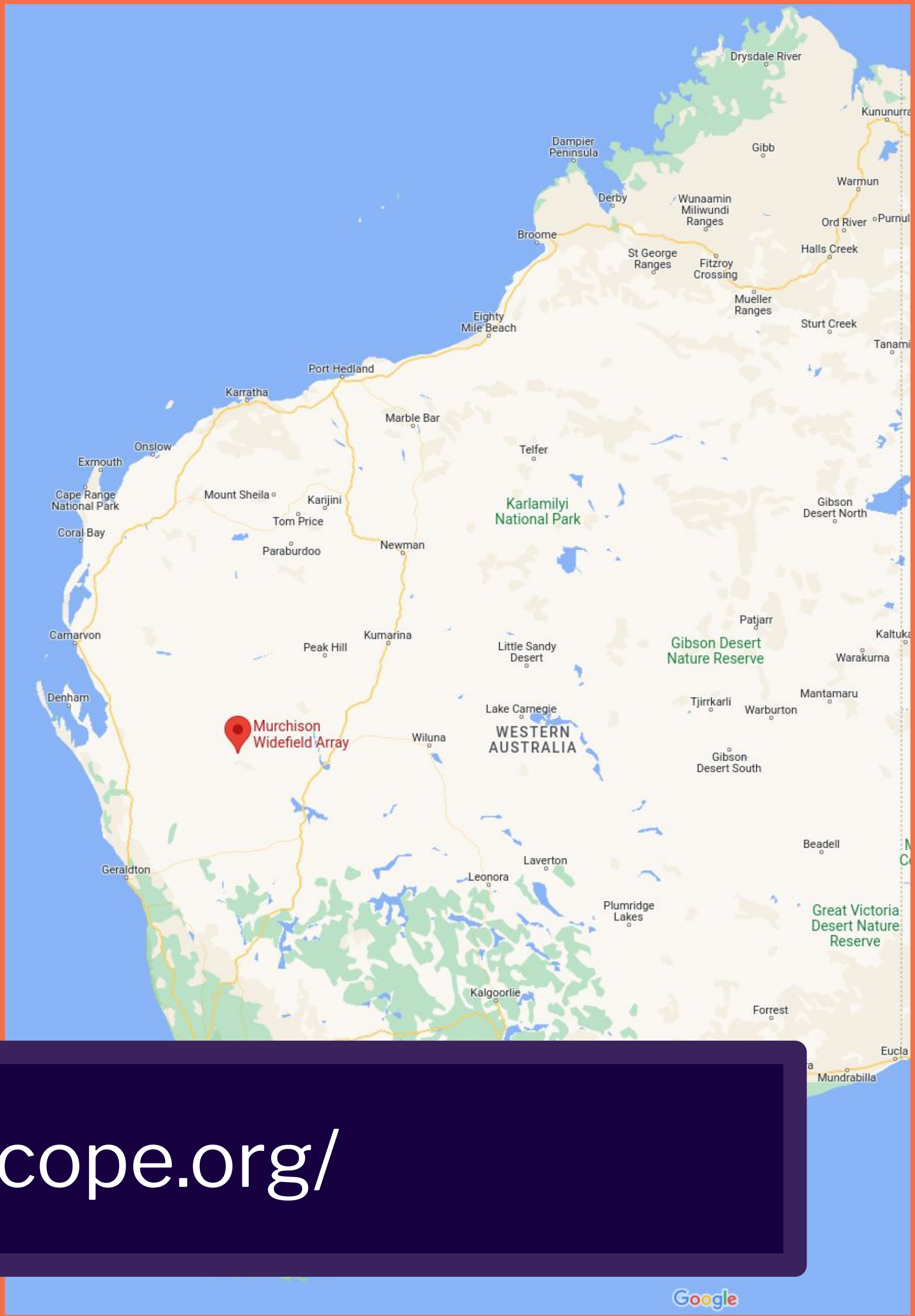

50 metres


Drysdale River
Kununurra
Dampier Peninsula
Gibb
Derby
Wunaamin Miliwundi Ranges
Warmun
Broome
St George Ranges
Fitzroy Crossing
Ord River
Halls Creek
Mueller Ranges
Sturt Creek
Eighty Mile Beach
Port Hedland
Karratha
Marble Bar
Onslow
Exmouth
Telfer
Cape Range National Park
Mount Sheila
Karijini
Tom Price
Karlamilyi National Park
Coral Bay
Paraburdoo
Newman
Gibson Desert North
Carnarvon
Kumarina
Peak Hill
Little Sandy Desert
Patjarr
Gibson Desert Nature Reserve
Warakurna
Denham
Murchison Widefield Array
Wiluna
Lake Carnegie
WESTERN AUSTRALIA
Tjirrkarli
Warburton
Mantamaru
Gibson Desert South
Geraldton
Laverton
Leonora
Beadell
Plumridge Lakes
Great Victoria Desert Nature Reserve
Kalgoorlie
Forrest
Perth
Rockingham
Norseman
Madura
Mundrabilla
Eucla
Margaret River
Esperance
Albany
Google

The MWA







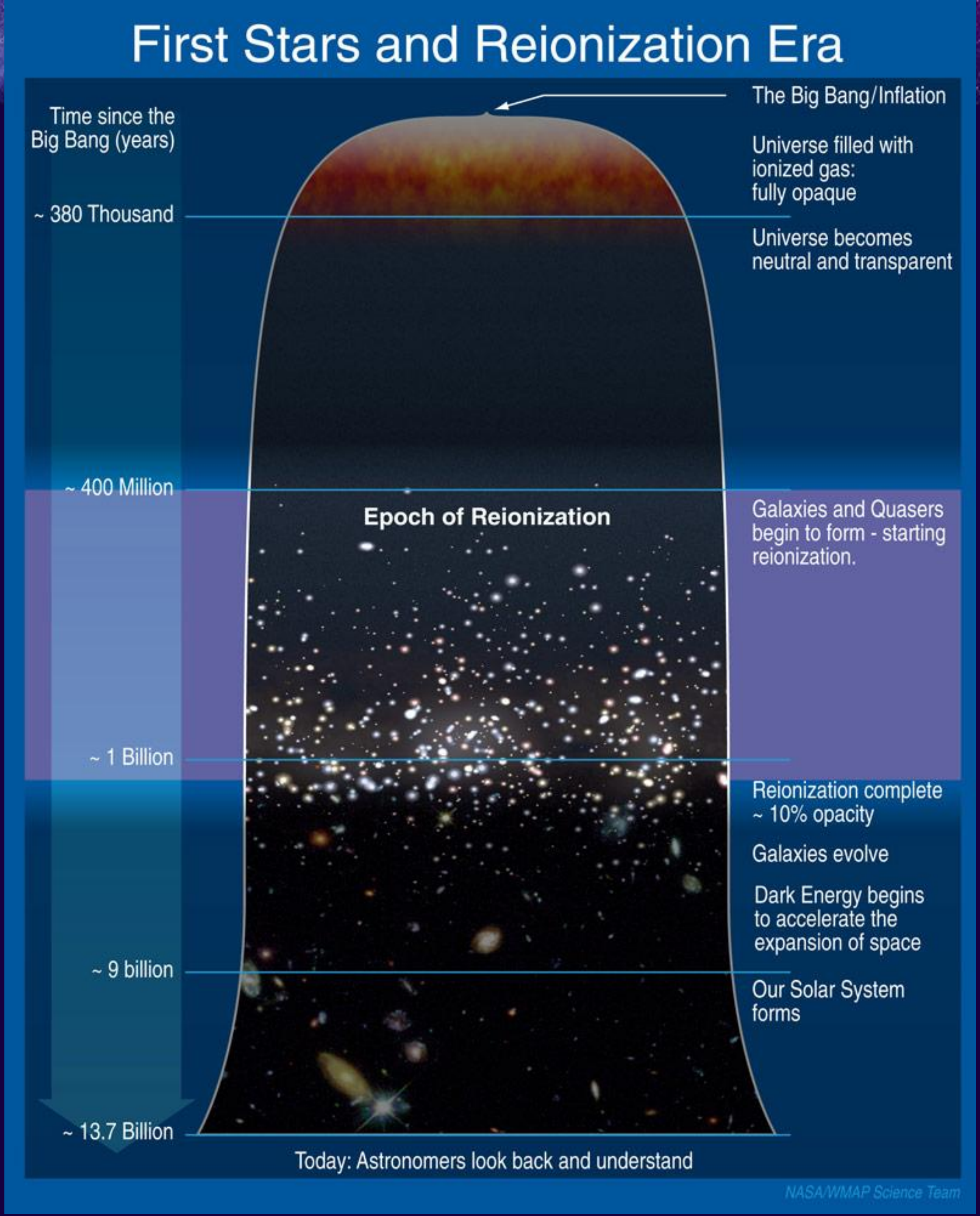
https://www.mwatelescope.org/

The EoR

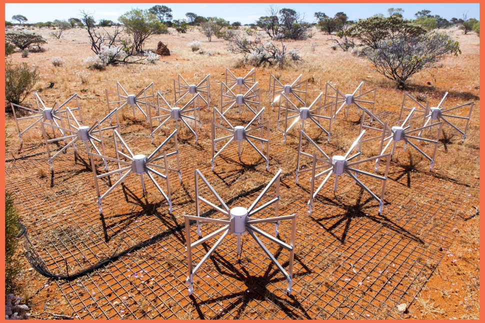
- Epoch of Reionisation (EoR)
  - First stars and galaxies
  - One of many MWA science projects

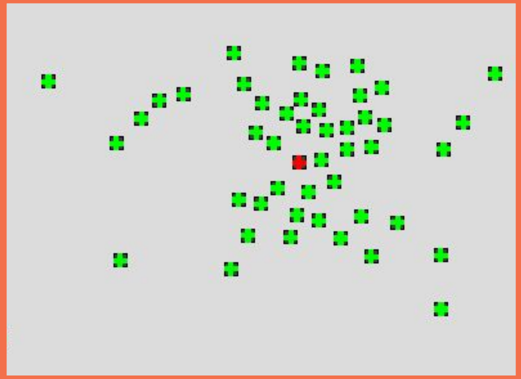
- Goal: Detect the EoR signature
  - Our approach: Neutral hydrogen

- Problems:
  - Extremely weak, distant signal
  - All other astronomical objects are “in the way”
  - Local radio interference
  - Earth’s ionosphere



The Data





antA	antB	ch0.xx	ch0.xy	ch0.yx	ch0.yy	ch1.xx	ch1.xy	ch1.yx	ch1.yy
0	0	r, i	r, i	r, i	r, i	r, i	r, i	r, i	r, i
0	1	r, i	r, i	r, i	r, i	r, i	r, i	r, i	r, i
0	2	r, i	r, i	r, i	r, i	r, i	r, i	r, i	r, i
0	3	r, i	r, i	r, i	r, i	r, i	r, i	r, i	r, i
1	1	r, i	r, i	r, i	r, i	r, i	r, i	r, i	r, i
1	2	r, i	r, i	r, i	r, i	r, i	r, i	r, i	r, i
1	3	r, i	r, i	r, i	r, i	r, i	r, i	r, i	r, i
2	2	r, i	r, i	r, i	r, i	r, i	r, i	r, i	r, i
2	3	r, i	r, i	r, i	r, i	r, i	r, i	r, i	r, i
3	3	r, i	r, i	r, i	r, i	r, i	r, i	r, i	r, i

- All MWA data is stored in 3 dims:
  - time
  - baseline
  - channel
- A unit of data is 8 f32s

- A typical observation has:
  - 50 times (~ 2 minutes)
  - 8128 baselines
  - 768 channels
  - ~10 GB

- EoR needs ~ 1000 hours!

Rust solution: hyperdrive

- Previous MWA code was written in C, C++ & Python
  - All slow
  - All difficult to install
    - Despite supercomputers!
  - All onerous to run
  - Somewhat difficult to use
  - Poorly documented

- Rust rewrite: hyperdrive (search: mwa_hyperdrive)
  - Extremely easy to use
  - Easy to install
  - (Somewhat) well documented
  - Optionally uses a CUDA/HIP GPU
  - … and very fast

https://github.com/MWATelescope/mwa_hyperdrive

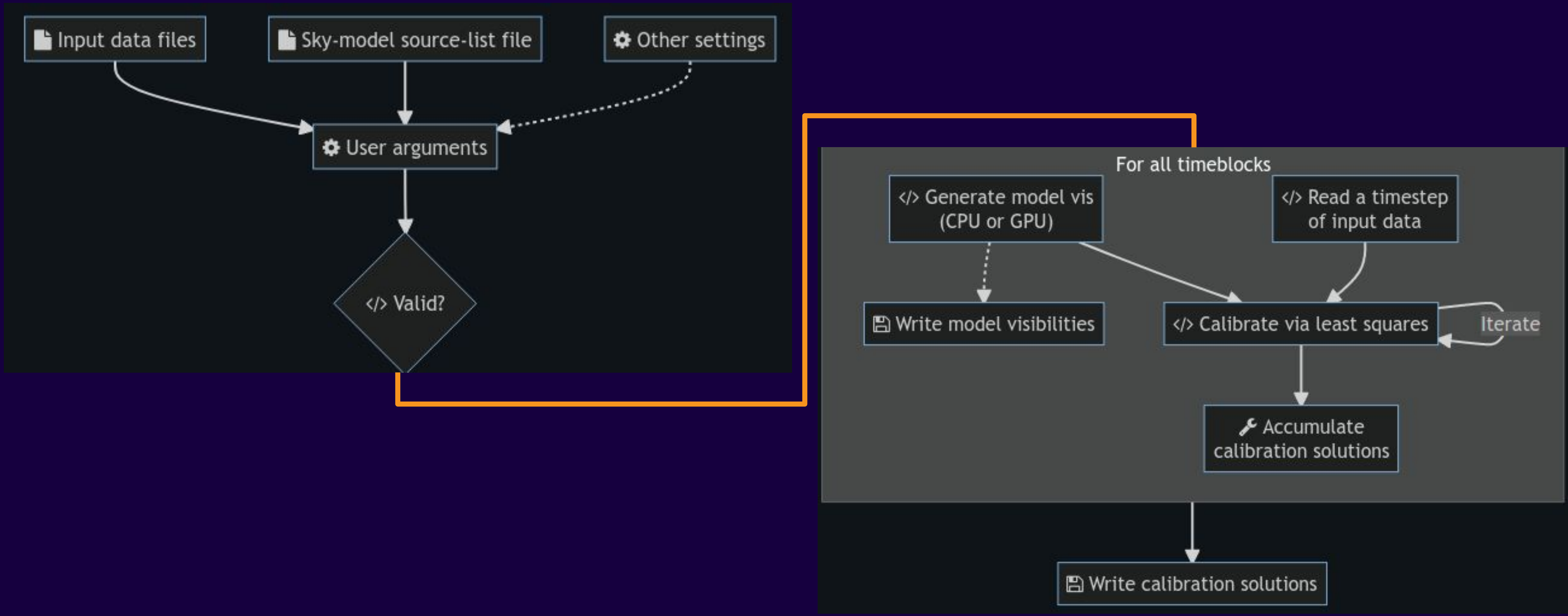
Rust solution: hyperdrive

- Extremely useful crates:
  - ndarray (+ approx when testing)
  - hifitime
  - clap
  - bindgen (C dependencies + DIY GPU code)
  - rayon
  - crossbeam family

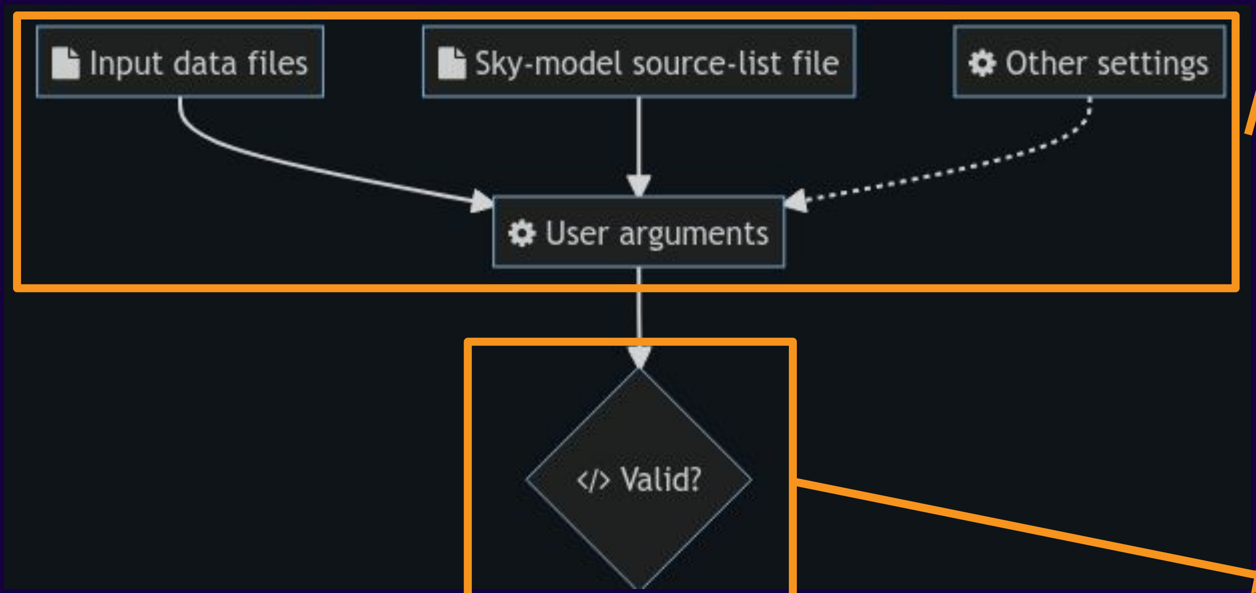
- Useful crates:
  - num-complex
  - thiserror
  - serde
  - indexmap
  - vec1
  - plotters
  - pyo3

https://github.com/MWATelescope/mwa_hyperdrive

Programming approach



Programming approac




```rust
#[derive(Parser, Debug, Clone, Default, Serialize, Deserialize)]
struct DiCalCliArgs {
    #[clap(short='o', long="outputs", multiple_values(true), help = DI_SOLS_OUTPUT
    solutions: Option<Vec<PathBuf>>,

    /// The number of timesteps to average together during calibration. Also
    /// supports a target time resolution (e.g. 8s). If this is 0, then all data
    /// are averaged together. Default: 0. e.g. If this variable is 4, then we
    /// produce calibration solutions in timeblocks with up to 4 timesteps each.
    /// If the variable is instead 4s, then each timeblock contains up to 4s
    /// worth of data.
    #[clap(short, long, help_heading = "CALIBRATION")]
    timesteps_per_timeblock: Option<String>,
```

```rust
/// Parse the arguments into parameters ready for calibration.
fn parse(self) -> Result<DiCalParams, HyperdriveError> {
    debug!("{:#?}", self);

    let DiCalArgs {
        args_file: _,
        data_args,
        srclist_args,
        model_args,
        beam_args,
        calibration_args,
    } = self;

    let input_vis_params = data_args.parse("DI calibrating")?;
    let obs_context = input_vis_params.get_obs_context();
    let total_num_tiles = input_vis_params.get_total_num_tiles();
```

Programming approach

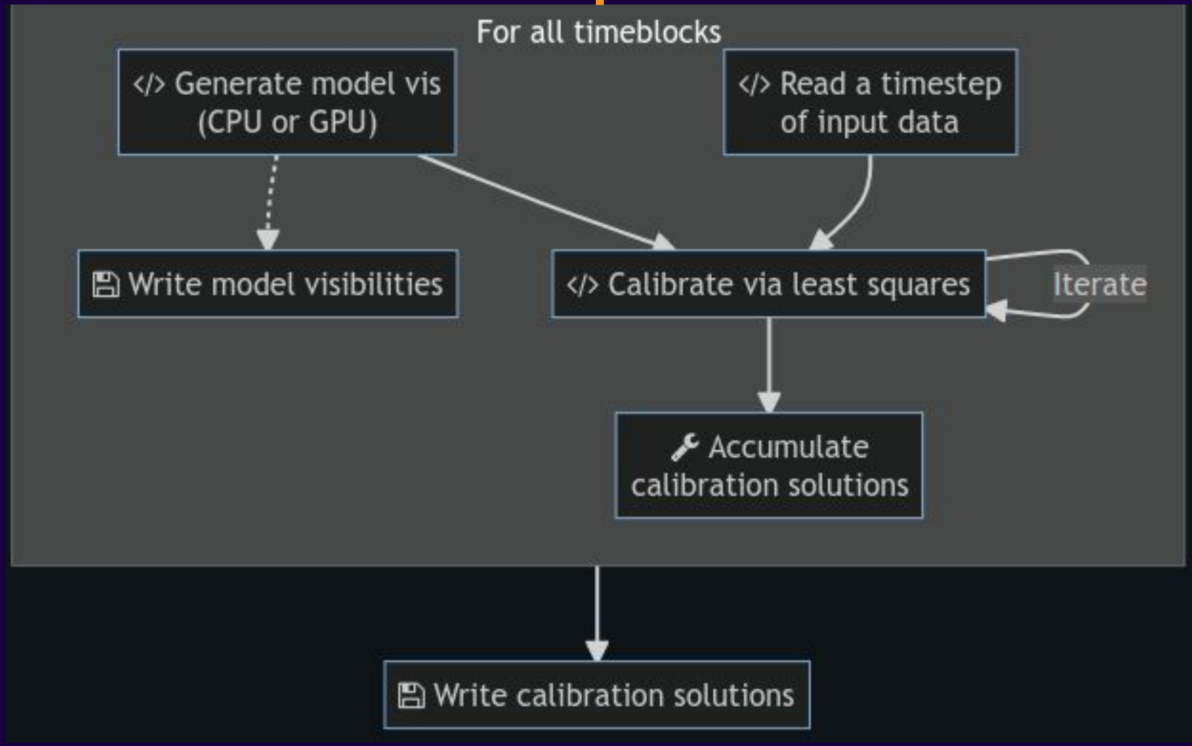
```rust
/// Use the [`DiCalParams`] to perform calibration and obtain solutions.
pub(crate) fn calibrate(&self) -> Result<CalibrationSolutions, super::DiCalibrateError> {
    // TODO: Fix.
    if self.freq_average_factor > 1 {
        panic!("Frequency averaging isn't working right now. Sorry!");
    }

    let CalVis {
        vis_data,
        vis_weights,
        vis_model,
    } = get_cal_vis(self, !self.no_progress_bars)?;
    assert_eq!(vis_weights.len_of(Axis(1)), self.baseline_weights.len());
```

```rust
// Use a variable to track whether any threads have an issue.
let error = AtomicCell::new(false);
info!("Reading input data and sky modelling");
let scoped_threads_result = thread::scope(|scope| {
```

```rust
// Input visibility-data reading thread.
let data_handle = scope.spawn(|_| {
```

```rust
// Sky-model generation thread.
let model_handle = scope.spawn(|_| {
    defer_on_unwind! { error.store(true); }
```



Programming approach

- User command-line args are ingested
  - Everything is optional; let parsers determine what's needed
- Args are passed to parsers to create "params"
  - Everything is validated while parsed
    - Files readable/writable? etc.
  - Error messages are very detailed here
- Param structs are used to actually do work
  - Steps and dependencies are done on separate threads
    - Keeps memory low and disk IO/CPU high
  - Failure is less likely → performance is better

- Advantages
  - Modular
  - Correct
- Disadvantages
  - Test code can have lots of "ceremony" to get required types
  - (Probably) harsh on the compiler
  - "Interesting" code is "deep"; compile times mean it is slow to get feedback on new code

ASTRO 3D

Programming approach: Use CPU or GPU

```rust
/// An object that simulates sky-model visibilities.
pub(crate) trait SkyModeller<'a> {
    /// Generate sky-model visibilities for a single timestep. The [UVW]
    /// coordinates used in generating the visibilities are returned.
    ///
    /// `vis_model_slice`: A mutable view into an `ndarray`. Rather than
    /// returning an array from this function, modelled visibilities are written
    /// into this array. This slice *must* have dimensions `[n1][n2]`, where
    /// `n1` is number of unflagged cross correlation baselines and `n2` is the
    /// number of unflagged frequencies.
    ///
    /// `timestamp`: The [hifitime::Epoch] struct used to determine what this
    /// timestep corresponds to.
    ///
    /// # Errors
    ///
    /// This function will return an error if there was a problem with
    /// beam-response calculation.
    fn model_timestep(
        &self,
        vis_model_slice: ArrayViewMut2<Jones<f32>>,
        timestamp: Epoch,
    ) -> Result<Vec<UVW>, BeamError>;
```

CPU

```rust
// Iterate over the unflagged baseline axis.
vis_model_slice
    .outer_iter_mut()
    .into_par_iter()
    .zip(uvws.par_iter())
    .enumerate()
    .for_each(|(i_baseline, (mut model_bl_axis, uvw_metres))| {
        let (i_tile1, i_tile2) = self.unflagged_baseline_to_tile_map[&i_baseline];

        // Unflagged fine-channel axis.
        model_bl_axis
            .iter_mut()
            .zip(fds.outer_iter())
            .zip(self.unflagged_fine_chan_freqs)
            .zip(beam_responses.slice(s![i_tile1, .., ..]).outer_iter())
            .zip(beam_responses.slice(s![i_tile2, .., ..]).outer_iter())
            .for_each(
                |((((model_vis, comp_fds), freq), tile1_beam), tile2_beam)| {
                    // Divide UVW by lambda to make UVW dimensionless.
                    let uvw = *uvw_metres * *freq / VEL_C;
```

Programming approach: Use CPU or GPU

```rust
/// An object that simulates sky-model visibilities.
pub(crate) trait SkyModeller<'a> {
    /// Generate sky-model visibilities for a single timestep. The [UVW]
    /// coordinates used in generating the visibilities are returned.
    ///
    /// `vis_model_slice`: A mutable view into an `ndarray`. Rather than
    /// returning an array from this function, modelled visibilities are written
    /// into this array. This slice *must* have dimensions `[n1][n2]`, where
    /// `n1` is number of unflagged cross correlation baselines and `n2` is the
    /// number of unflagged frequencies.
    ///
    /// `timestamp`: The [hifitime::Epoch] struct used to determine what this
    /// timestep corresponds to.
    ///
    /// # Errors
    ///
    /// This function will return an error if there was a problem with
    /// beam-response calculation.
    fn model_timestep(
        &self,
        vis_model_slice: ArrayViewMut2<Jones<f32>>,
        timestamp: Epoch,
    ) -> Result<Vec<UVW>, BeamError>;
```

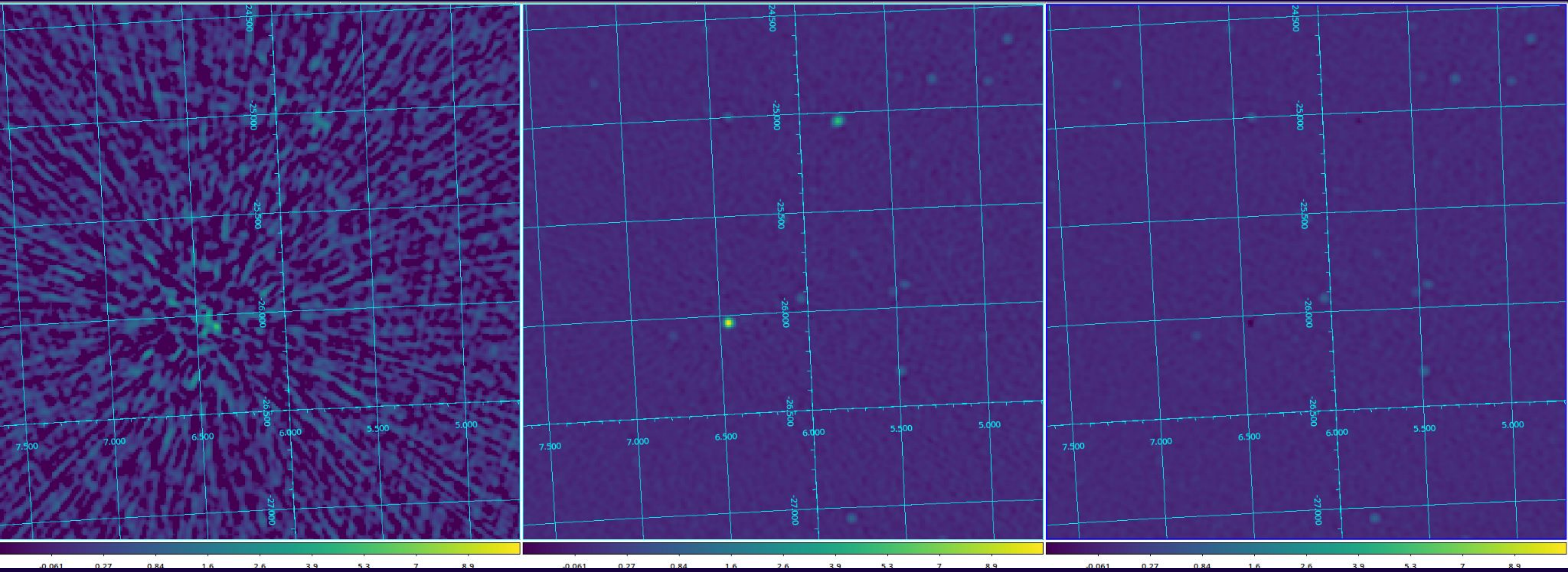
GPU

```rust
let cuda_status = cuda::model_points(
    &cuda::Points {
        num_power_law_points: self.point_power_law_radecs.len(),
        power_law_lmns: self.point_power_law_lmns.get_mut(),
        power_law_fds: self.point_power_law_fds.get_mut(),
        power_law_sis: self.point_power_law_sis.get_mut(),
```

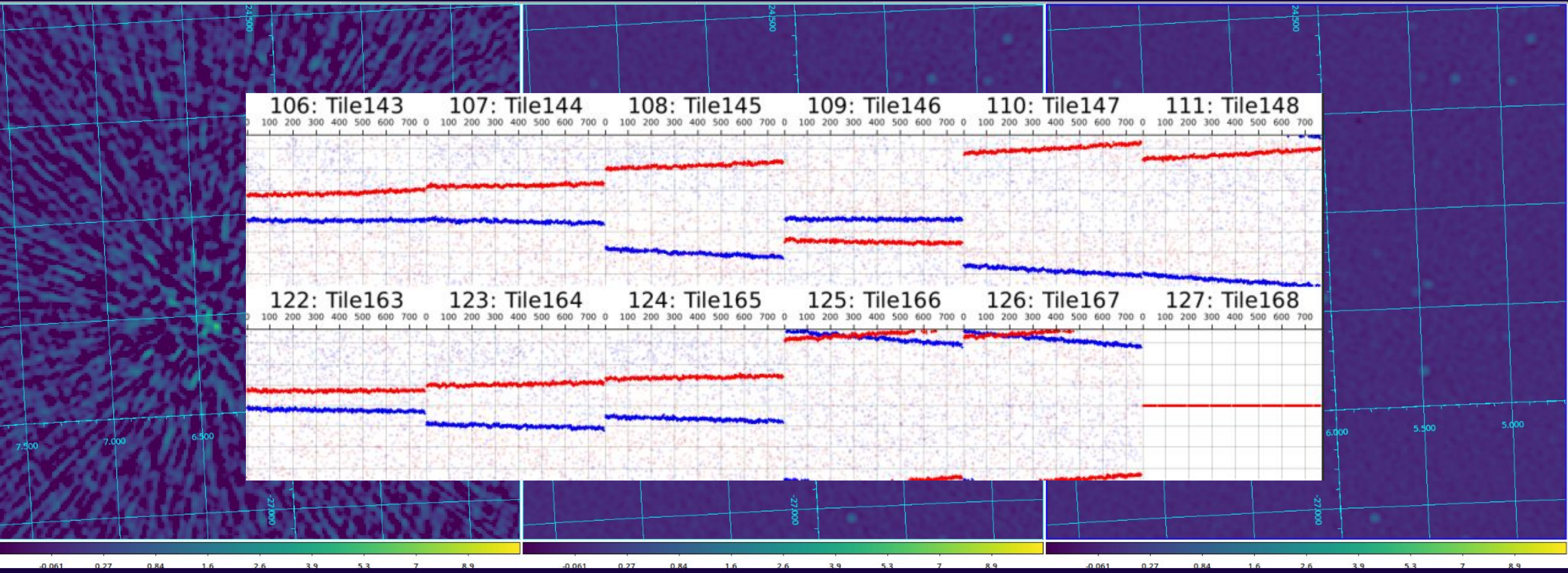
```c
extern "C" int model_points(const Points *comps, const Addresses *a, const
    dim3 gridDim, blockDim;
    if (a->num_unique_beam_freqs == 0) {
        // Thread blocks are distributed by visibility (one visibility per
        // frequency and baseline).
        blockDim.x = 512;
        blockDim.y = 1;
        gridDim.x = (int)ceil((double)a->num_vis / (double)blockDim.x);
        gridDim.y = 1;

        model_points_kernel<<<gridDim, blockDim>>>(a->num_freqs, a->num_vis
        cudaCheck(cudaPeekAtLastError());
```

(Brief) MWA EoR results

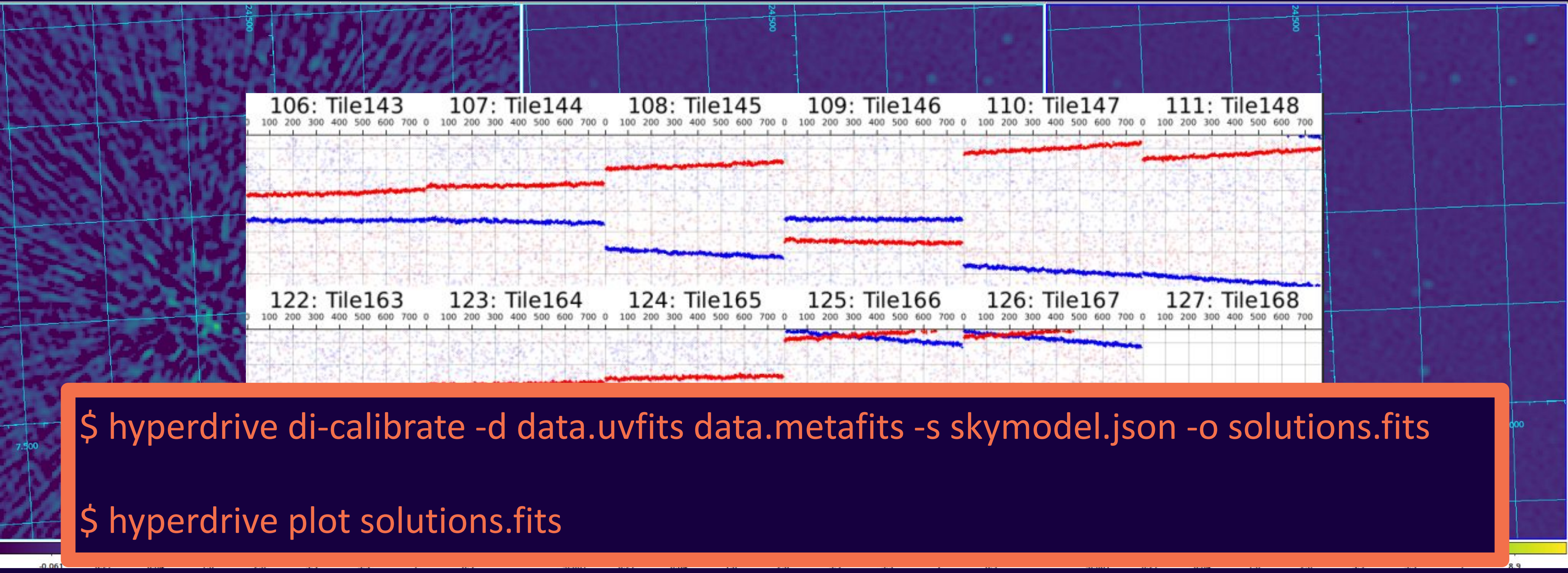


(Brief) MWA EoR results



ASTRO 3D

(Brief) MWA EoR results




```
$ hyperdrive di-calibrate -d data.uvfits data.metafits -s skymodel.json -o solutions.fits

$ hyperdrive plot solutions.fits
```

Using Rust has dramatically eased our:

- computational load,
- capacity to make changes quickly,
- maintenance burden, and
- ability to deploy/install on new machines.

But most importantly, Rust has enabled me to allow researchers to get on with their research.

I therefore strongly recommend using Rust for reliable, fast code.

Thanks!