ICRAR

International Centre for Radio Astronomy Research

The Bifrost Stream Processing Framework

Danny Price, Jayce Dowell, Christopher League
Ben Barsdell, Miles Cranmer, Lincoln Greenhill, Greg Taylor
+ other collaborators

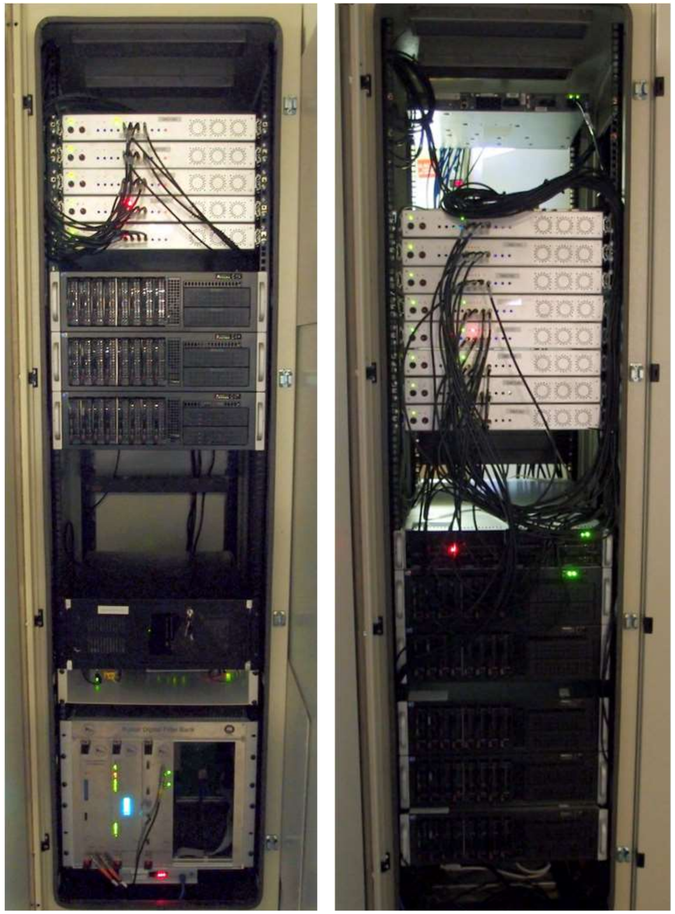
Danny Price
ICRAR / Curtin University
danny.price@curtin.edu.au

Curtin University

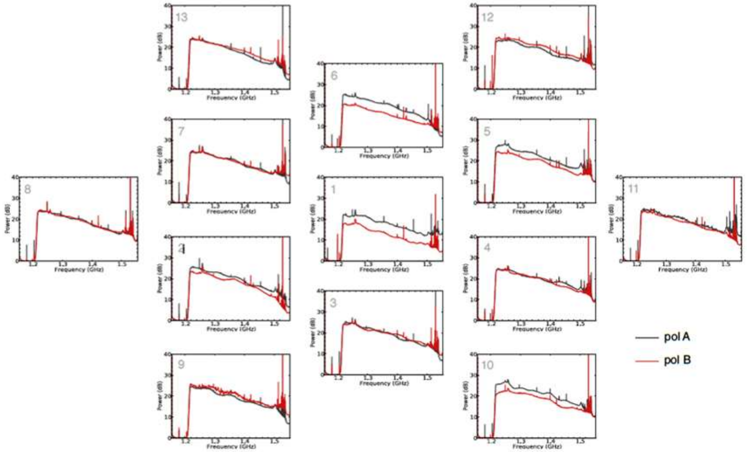
THE UNIVERSITY OF WESTERN AUSTRALIA

Government of Western Australia
Department of the Premier and Cabinet
Office of Science

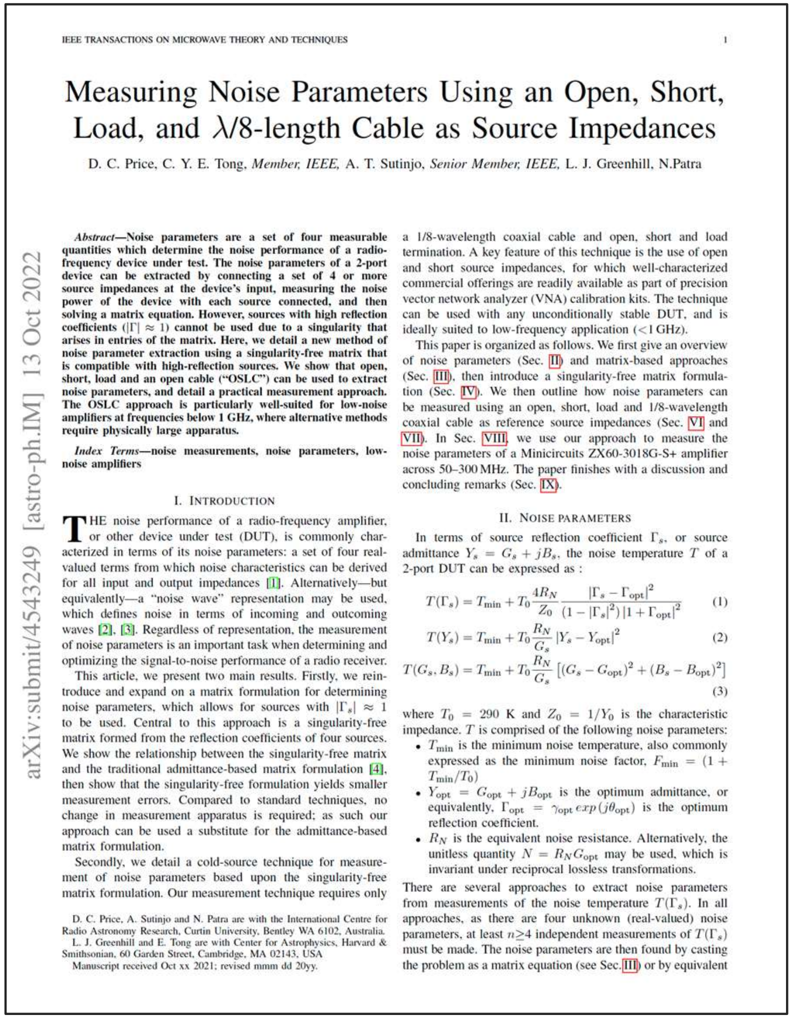
Sidenote: Multibeam receivers were difficult enough already!



Parkes HIPSR Racks



Sidenote: Measuring Noise Parameters



Measuring Noise Parameters Using an Open, Short, Load, and $\lambda/8$-length Cable as Source Impedances

D. C. Price, C. Y. E. Tong, *Member, IEEE,* A. T. Sutinjo, *Senior Member, IEEE,* L. J. Greenhill, N.Patra

***Abstract*—Noise parameters are a set of four measurable quantities which determine the noise performance of a radio-frequency device under test. The noise parameters of a 2-port device can be extracted by connecting a set of 4 or more source impedances at the device's input, measuring the noise power of the device with each source connected, and then solving a matrix equation. However, sources with high reflection coefficients ($|\Gamma| \approx 1$) cannot be used due to a singularity that arises in entries of the matrix. Here, we detail a new method of noise parameter extraction using a singularity-free matrix that is compatible with high-reflection sources. We show that open, short, load and an open cable ("OSLC") can be used to extract noise parameters, and detail a practical measurement approach. The OSLC approach is particularly well-suited for low-noise amplifiers at frequencies below 1 GHz, where alternative methods require physically large apparatus.**

***Index Terms*—noise measurements, noise parameters, low-noise amplifiers**

I. Introduction

THE noise performance of a radio-frequency amplifier, or other device under test (DUT), is commonly characterized in terms of its noise parameters: a set of four real-valued terms from which noise characteristics can be derived for all input and output impedances [1]. Alternatively—but equivalently—a "noise wave" representation may be used, which defines noise in terms of incoming and outcoming waves [2], [3]. Regardless of representation, the measurement of noise parameters is an important task when determining and optimizing the signal-to-noise performance of a radio receiver.

This article, we present two main results. Firstly, we reintroduce and expand on a matrix formulation for determining noise parameters, which allows for sources with $|\Gamma_s| \approx 1$ to be used. Central to this approach is a singularity-free matrix formed from the reflection coefficients of four sources. We show the relationship between the singularity-free matrix and the traditional admittance-based matrix formulation [4], then show that the singularity-free formulation yields smaller measurement errors. Compared to standard techniques, no change in measurement apparatus is required; as such our approach can be used a substitute for the admittance-based matrix formulation.

Secondly, we detail a cold-source technique for measurement of noise parameters based upon the singularity-free matrix formulation. Our measurement technique requires only a 1/8-wavelength coaxial cable and open, short and load termination. A key feature of this technique is the use of open and short source impedances, for which well-characterized commercial offerings are readily available as part of precision vector network analyzer (VNA) calibration kits. The technique can be used with any unconditionally stable DUT, and is ideally suited to low-frequency application (<1 GHz).

This paper is organized as follows. We first give an overview of noise parameters (Sec. II) and matrix-based approaches (Sec. III), then introduce a singularity-free matrix formulation (Sec. IV). We then outline how noise parameters can be measured using an open, short, load and 1/8-wavelength coaxial cable as reference source impedances (Sec. VI and VII). In Sec. VIII, we use our approach to measure the noise parameters of a Minicircuits ZX60-3018G-S+ amplifier across 50–300 MHz. The paper finishes with a discussion and concluding remarks (Sec. IX).

II. Noise parameters

In terms of source reflection coefficient Γ_s, or source admittance $Y_s = G_s + jB_s$, the noise temperature T of a 2-port DUT can be expressed as :

$$T(\Gamma_s) = T_{\min} + T_0 \frac{4R_N}{Z_0} \frac{|\Gamma_s - \Gamma_{\mathrm{opt}}|^2}{(1-|\Gamma_s|^2)\,|1+\Gamma_{\mathrm{opt}}|^2} \quad (1)$$

$$T(Y_s) = T_{\min} + T_0 \frac{R_N}{G_s} |Y_s - Y_{\mathrm{opt}}|^2 \quad (2)$$

$$T(G_s, B_s) = T_{\min} + T_0 \frac{R_N}{G_s} \left[(G_s - G_{\mathrm{opt}})^2 + (B_s - B_{\mathrm{opt}})^2\right] \quad (3)$$

where $T_0 = 290$ K and $Z_0 = 1/Y_0$ is the characteristic impedance. T is comprised of the following noise parameters:

- $T_{\min}$ is the minimum noise temperature, also commonly expressed as the minimum noise factor, $F_{\min} = (1 + T_{\min}/T_0)$
- $Y_{\mathrm{opt}} = G_{\mathrm{opt}} + jB_{\mathrm{opt}}$ is the optimum admittance, or equivalently, $\Gamma_{\mathrm{opt}} = \gamma_{\mathrm{opt}} exp(j\theta_{\mathrm{opt}})$ is the optimum reflection coefficient.
- R_N is the equivalent noise resistance. Alternatively, the unitless quantity $N = R_N G_{\mathrm{opt}}$ may be used, which is invariant under reciprocal lossless transformations.

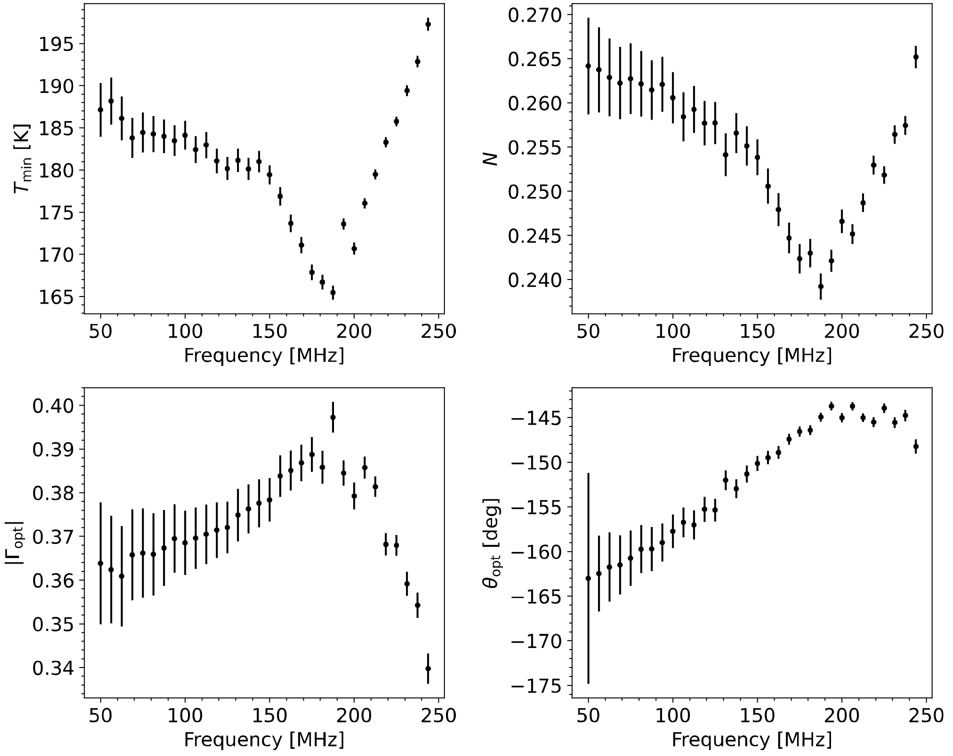
There are several approaches to extract noise parameters from measurements of the noise temperature $T(\Gamma_s)$. In all approaches, as there are four unknown (real-valued) noise parameters, at least $n \geq 4$ independent measurements of $T(\Gamma_s)$ must be made. The noise parameters are then found by casting the problem as a matrix equation (see Sec. III) or by equivalent

D. C. Price, A. Sutinjo and N. Patra are with the International Centre for Radio Astronomy Research, Curtin University, Bentley WA 6102, Australia.
L. J. Greenhill and E. Tong are with Center for Astrophysics, Harvard & Smithsonian, 60 Garden Street, Cambridge, MA 02143, USA
Manuscript received Oct xx 2021; revised mmm dd 20yy.

- Noise parameters are a complete description of amplifier noise performance
  - Contain more information than standard Y-factor measurement, but tricky to measure.
- An LNA will have different noise performance when connected to an antenna than as measured with a Y-factor measurement.
- Our simple (and cheap!) method for measuring noise parameters accepted to IEEE transactions on Microwave Theory & Techniques.
- Preprint now available on arXiv: https://arxiv.org/abs/2210.07080

Noise Parameters

- The noise performance of a device under test (DUT) can be described by four parameters:
  - T_{min} minimum noise temperature.
  - Γ_{opt} optimal reflection coefficient (complex so counts as two params).
  - N equivalent noise resistance.
- For lowest noise temperature $\mathrm{T} = T_{\text{min}}$, we want to connect our DUT to a source with $\Gamma = \Gamma_{\text{opt}}$.

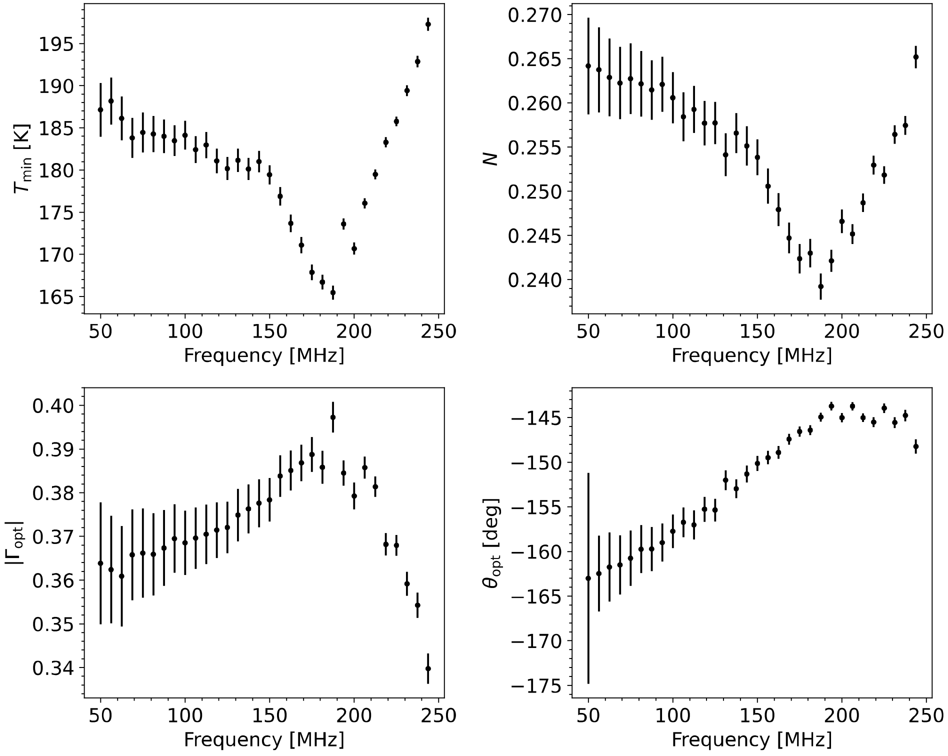


Noise Parameters

- The noise performance of a device under test (DUT) can be described by four parameters:

    - T_{min} minimum noise temperature.
    - Γ_{opt} optimal reflection coefficient (complex so counts as two params).
    - N equivalent noise resistance.

- For lowest noise temperature $\mathrm{T} = T_{\mathrm{min}}$, we want to connect our DUT to a source with $\Gamma = \Gamma_{\mathrm{opt}}$.



Questions for CryoPAFs:
Q1. How to modify for cryogenic application?
Q2. How to measure differential / multiport DUTs?

Part 1: Introduction

“Odin on a laptop” by Midjourney AI

What is stream processing?

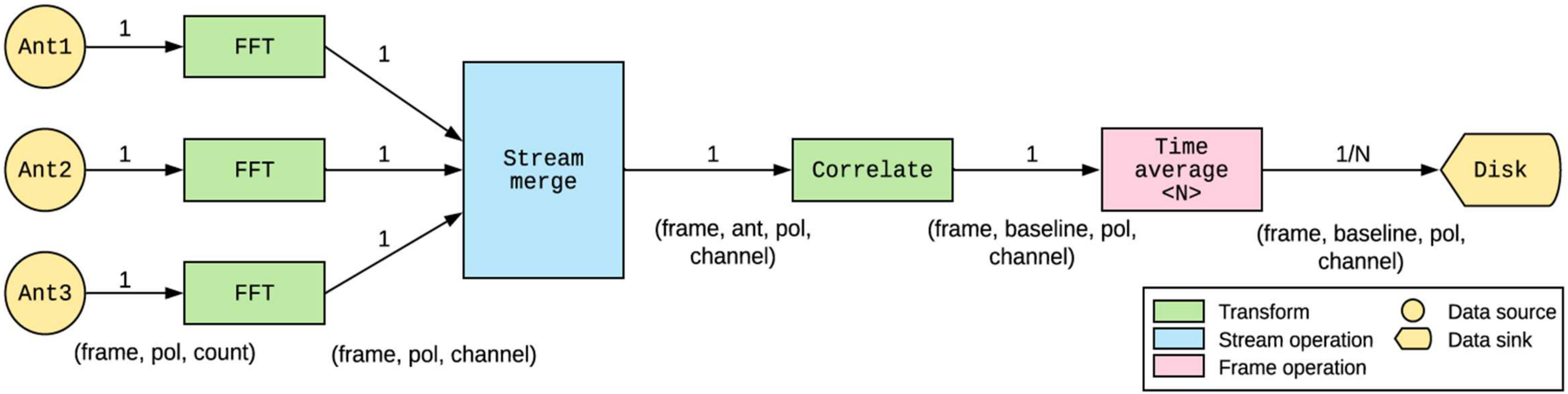
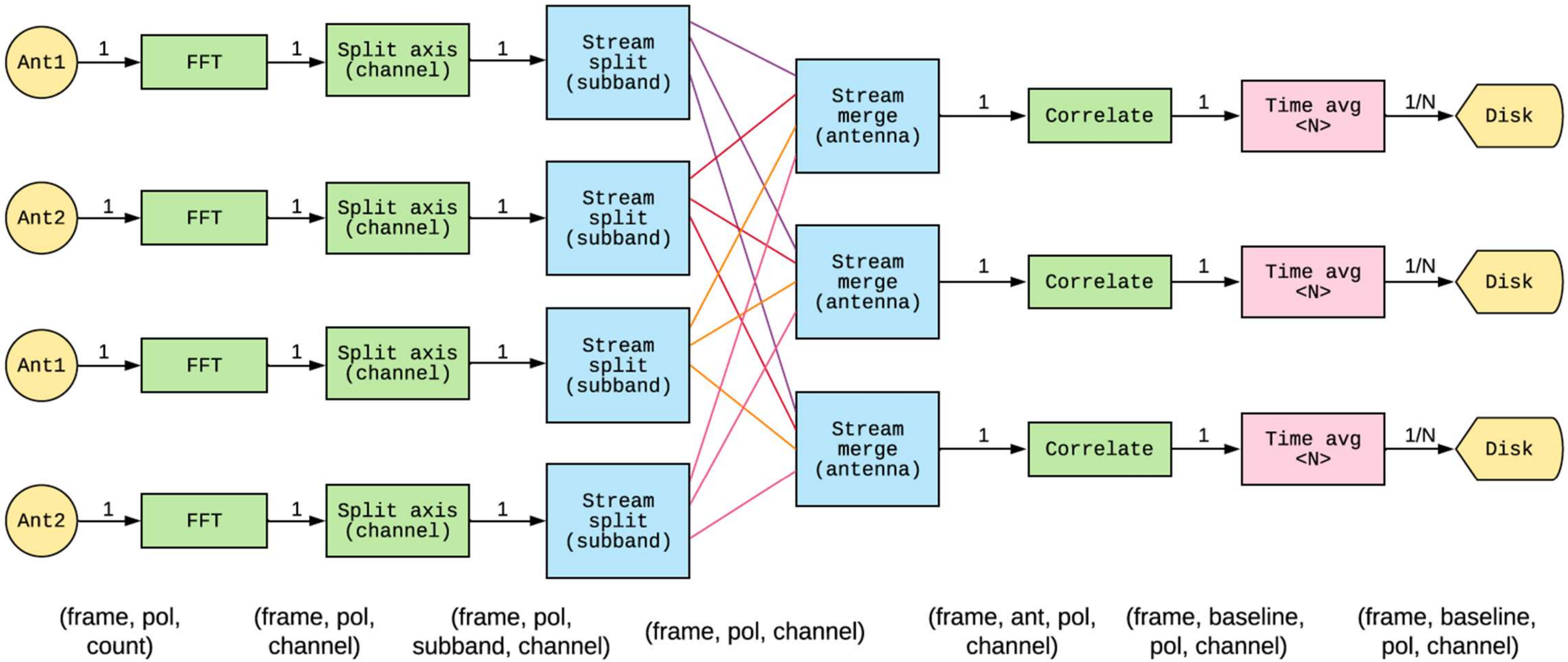


Figure 2: Simple example of a pipeline to cross-correlate three antennas, using data streams. Each antenna outputs a data stream, and a Fast Fourier Transform (FFT) is applied to each to form channels. The three streams are merged together, and then cross-correlation is applied. Multiple frames are then averaged together, before being written to disk. The number above the arrow represents the frame rate; frame dimensions are shown in brackets below the streams.

A more complex example



Data rates in astronomy stream processing

Table 2: Aggregate data rates after digitization for selected radio telescopes.

Telescope	F_s (MHz)	N_{bits}	N_{ant}	N_{rxb}	N_{pol}	N_{sub}	DR (Tb/s)	Reference
ASKAP	608	12	36	188	2	1	98.8	A
CHIME	800	8	1024	1	2	1	13.1	B
ALMA	4000	3	64	1	2	4	6.1	C
MeerKAT	1712	10	64	1	2	1	2.2	D
SMA	4000	8	8	1	1	4	2.0	E
JVLA	2048	8	26	1	2	2	1.7	F

[A] Schinckel et al. (2012)
[B] CHIME/FRB Collaboration et al. (2018); Bandura et al. (2016a)
[C] Baudry and Webber (2011)
[D] Jonas (2009)
[E] Primiani et al. (2016)
[F] Perley et al. (2011)

Table 3: Data ingest rates (per server) for selected second-stage signal processing instruments.

Instrument	Ingest rate (Gb/s/server)	Packet size (B)	Capture method	Data pipeline	Ref.
HERA	68.0	4608	ibverbs	HASHPIPE	A
TRAPUM	56.0	1024	ibverbs/SPEAD2	PSRDADA	B
CHIME	25.6	8592	DPDK	kotekan	C
Parkes UWL	24.8	8272	ibverbs	PSRDADA	D
LEDA	21.4	7008	socket	PSRDADA	E

[A] DeBoer et al. (2017), J. Hickish (personal comms)
[B] Stappers and Kramer (2016), E. Barr (personal comms)
[C] Recnik et al. (2015); CHIME/FRB Collaboration et al. (2018)
[D] Hobbs et al. (2019)
[E] Kocz et al. (2015)

Tables from “Stream processing in radio astronomy”,
DC Price, Chapter 4, Big Data in Astronomy, Elsevier (2020)

Data rates in astronomy stream processing

Table 2: Aggregate data rates after digitization for selected radio telescopes.

Telescope	F_s (MHz)	N_{bits}	N_{ant}	N_{rxb}	N_{pol}	N_{sub}	DR (Tb/s)	Reference
ASKAP	608	12	36	188	2	1	98.8	A
CHIME	800	8	1024	1	2	1	13.1	B
ALMA	4000	3	64	1	2	4	6.1	C
MeerKAT	1712	10	64	1	2	1	2.2	D
SMA	4000	8	8	1	1	4	2.0	E
JVLA	2048	8	26	1	2	2	1.7	F

[A] Schinckel et al. (2012)
[B] CHIME/FRB Collaboration et al. (2018); Bandura et al. (2016a)
[C] Baudry and Webber (2011)
[D] Jonas (2009)
[E] Primiani et al. (2016)
[F] Perley et al. (2011)

Table 3: Data ingest rates (per server) for selected second-stage signal processing instruments.

Instrument	Ingest rate (Gb/s/server)	Packet size (B)	Capture method	Data pipeline	Ref.
HERA	68.0	4608	ibverbs	HASHPIPE	A
TRAPUM	56.0	1024	ibverbs/SPEAD2	PSRDADA	B
CHIME	25.6	8592	DPDK	kotekan	C
Parkes UWL	24.8	8272	ibverbs	PSRDADA	D
LEDA	21.4	7008	socket	PSRDADA	E

[A] DeBoer et al. (2017), J. Hickish (personal comms)
[B] Stappers and Kramer (2016), E. Barr (personal comms)
[C] Recnik et al. (2015); CHIME/FRB Collaboration et al. (2018)
[D] Hobbs et al. (2019)
[E] Kocz et al. (2015)

EDD: 90+ Gb/s/NIC ibverbs (Ewan Barr, yesterday)

Tables from “Stream processing in radio astronomy”,
DC Price, Chapter 4, Big Data in Astronomy, Elsevier (2020)

Data rates in astronomy stream processing

Table 2: Aggregate data rates after digitization for selected radio telescopes.

Telescope	F_s (MHz)	N_{bits}	N_{ant}	N_{rxb}	N_{pol}	N_{sub}	DR (Tb/s)	Reference
ASKAP	608	12	36	188	2	1	98.8	A
CHIME	800	8	1024	1	2	1	13.1	B
ALMA	4000	3	64	1	2	4	6.1	C
MeerKAT	1712	10	64	1	2	1	2.2	D
SMA	4000	8	8	1	1	4	2.0	E
JVLA	2048	8	26	1	2	2	1.7	F

[A] Schinckel et al. (2012)
[B] CHIME/FRB Collaboration et al. (2018); Bandura et al. (2016a)
[C] Baudry and Webber (2011)
[D] Jonas (2009)
[E] Primiani et al. (2016)
[F] Perley et al. (2011)

Table 3: Data ingest rates (per server) for selected second-stage signal processing instruments.

Instrument	Ingest rate (Gb/s/server)	Packet size (B)	Capture method	Data pipeline	Ref.
HERA	68.0	4608	ibverbs	HASHPIPE	A
TRAPUM	56.0	1024	ibverbs/SPEAD2	PSRDADA	B
CHIME	25.6	8592	DPDK	kotekan	C
Parkes UWL	24.8	8272	ibverbs	PSRDADA	D
LEDA	21.4	7008	socket	PSRDADA	E

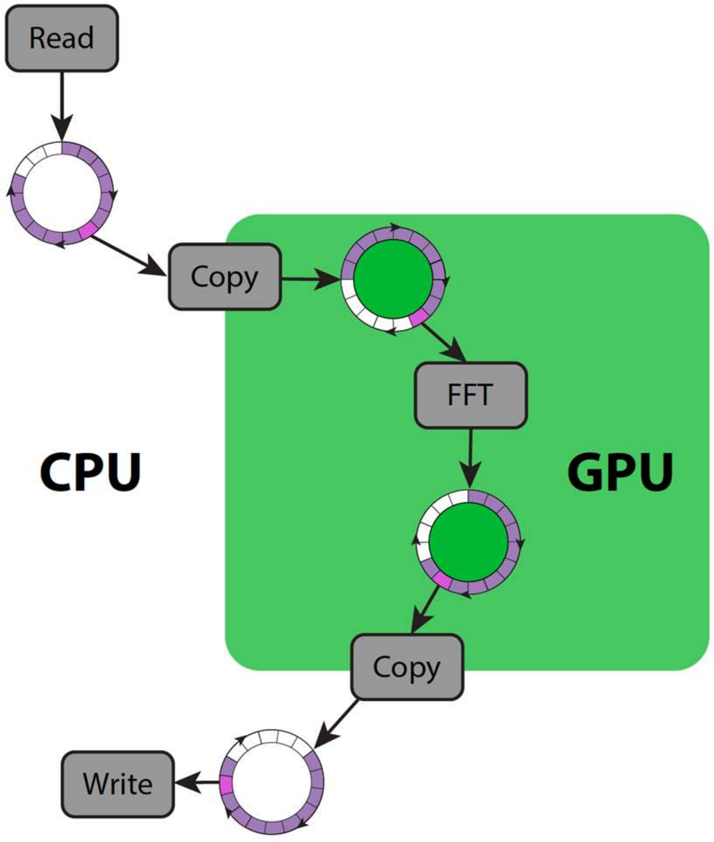
[A] DeBoer et al. (2017), J. Hickish (personal comms)
[B] Stappers and Kramer (2016), E. Barr (personal comms)
[C] Recnik et al. (2015); CHIME/FRB Collaboration et al. (2018)
[D] Hobbs et al. (2019)
[E] Kocz et al. (2015)

EDD: 90+ Gb/s/NIC ibverbs (Ewan Barr, yesterday)

Bifrost/LWA: ~70 Gb/s/NIC, ibverbs (Jayce Dowell)

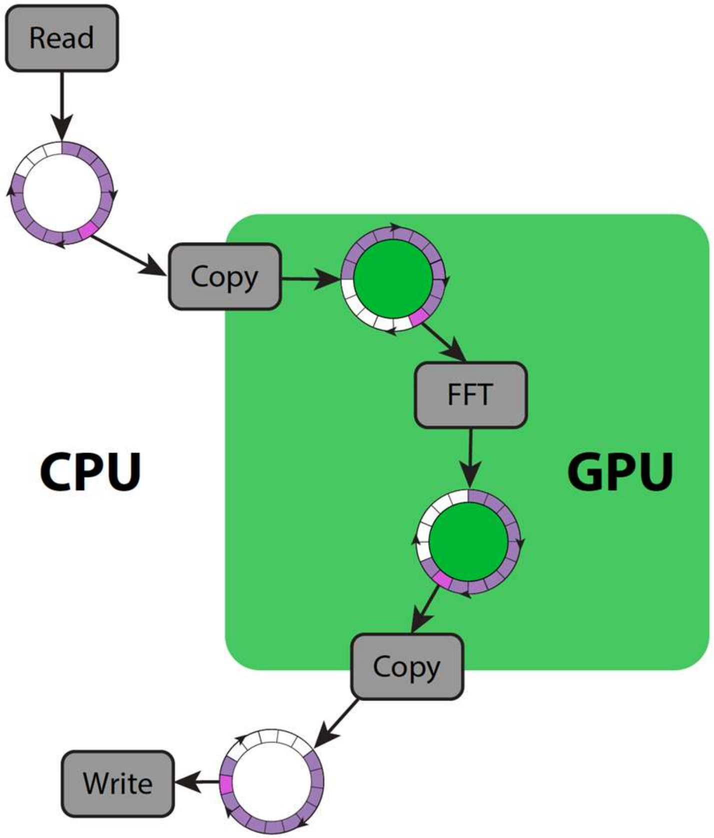
Tables from “Stream processing in radio astronomy”,
DC Price, Chapter 4, Big Data in Astronomy, Elsevier (2020)

Bifrost overview



- Bifrost is a C++/CUDA/Python framework for stream processing.
- Comes with reconfigurable and repurposable GPU/CPU blocks for common radio astronomy applications (e.g. FFT, FDMT, transpose, requantization)

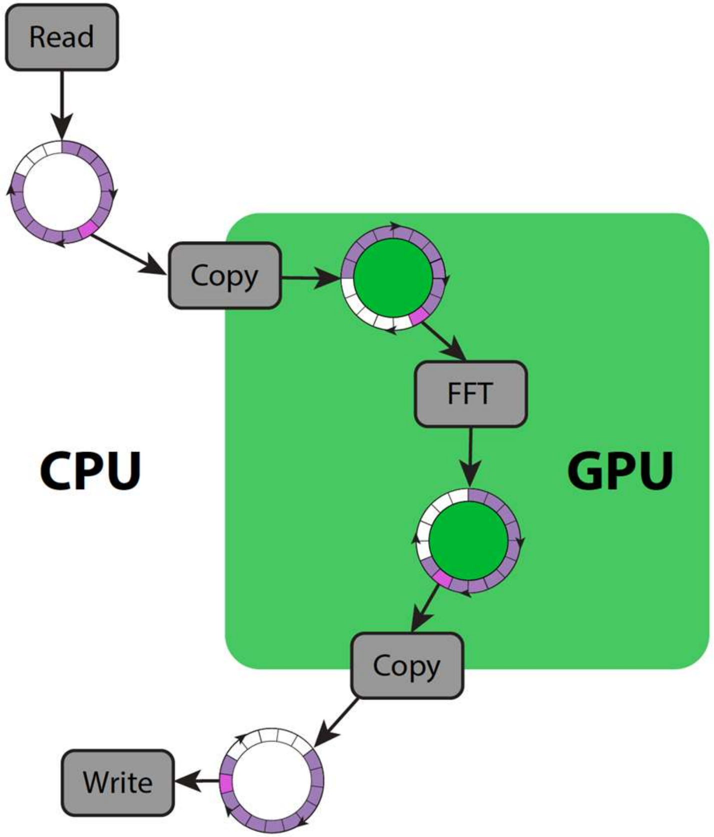
Bifrost overview



- Bifrost is a C++/CUDA/Python framework for stream processing.
- Comes with reconfigurable and repurposable GPU/CPU blocks for common radio astronomy applications (e.g. FFT, FDMT, transpose, requantization)
- Open-source license (BSD 3), code available on Github: https://github.com/ledatelescope/bifrost
- Plugin system bragr under development: https://github.com/bf_plugins

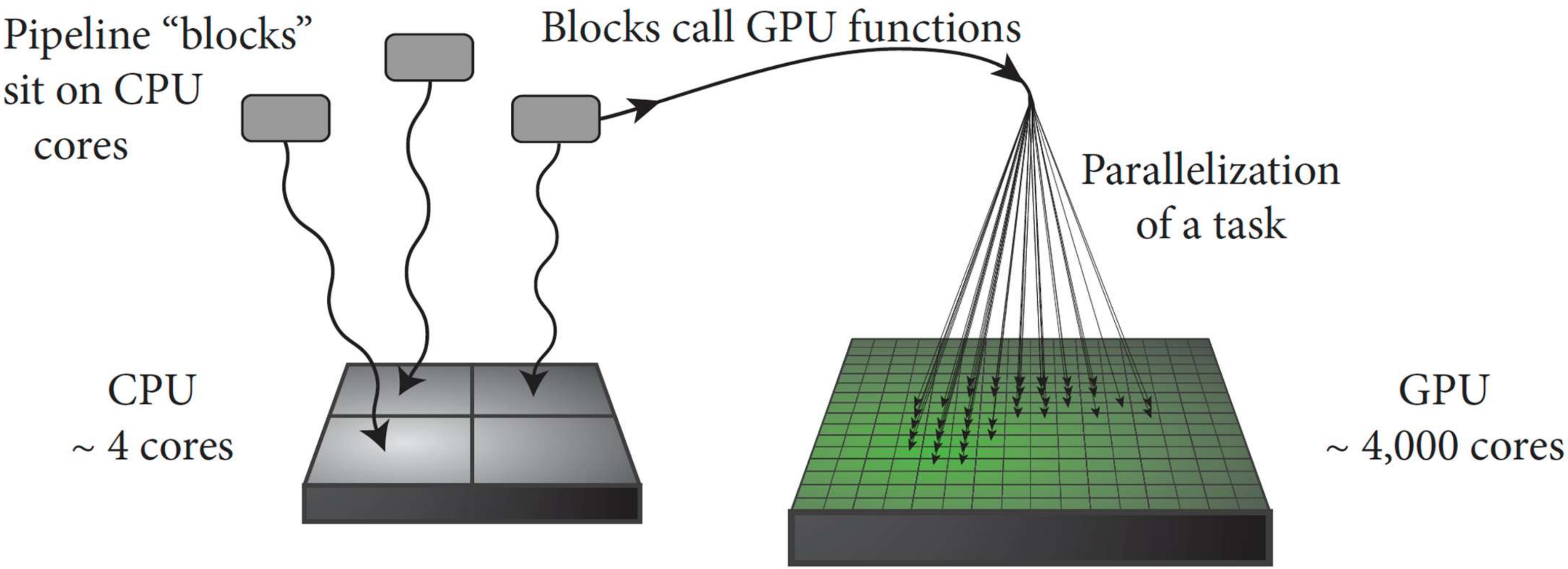
Bifrost summary paper:
Cranmer et al.(2017), *JAI*, 6, 1750007.

Bifrost overview

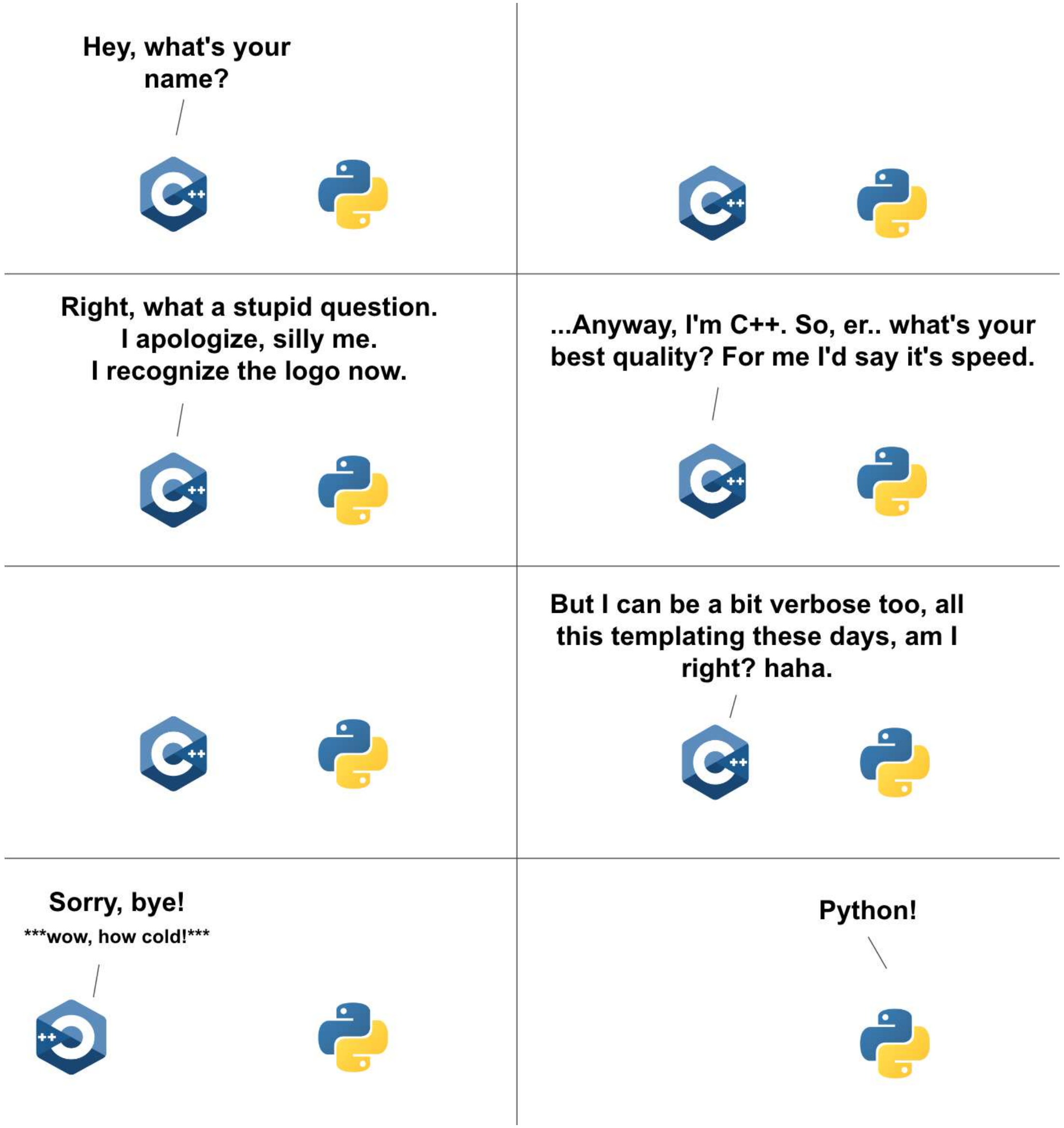


- Blocks: “atomic” data processing tasks. Each runs in its own thread.
- Ring buffers: First-in, first-out data queue. Contiguous pre-allocated memory on CPU/GPU (or cuda_managed) spaces.
- Pipelines: A combination of blocks and rings, connected as a directed acyclic graph, used to process data.

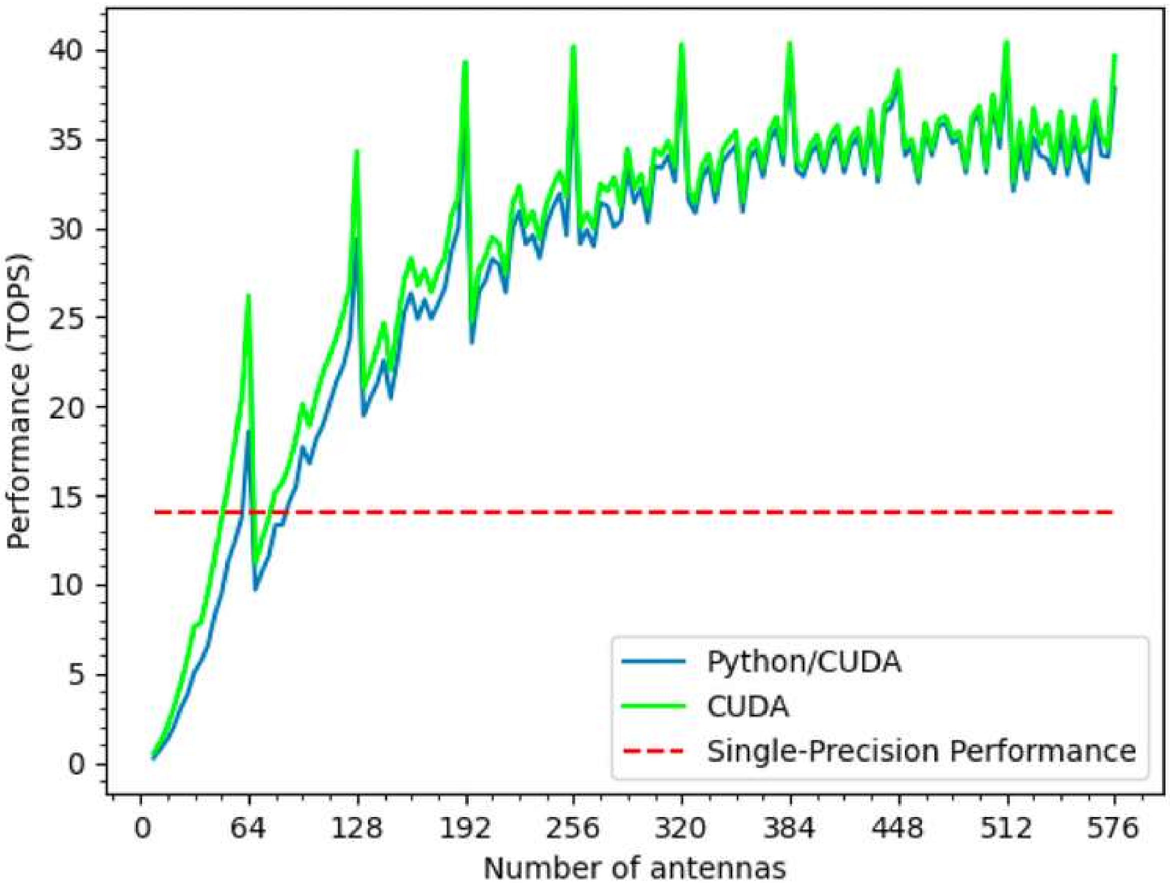
Bifrost overview



But isn't Python slow?



But isn't Python slow?



Bifrost-wrapped tensor core correlator performance (plot: Liam Ryan)

- Computationally-expensive stuff is done in C++/CUDA.
- Compiled C++ code is called via ctypes. Overhead is minimal if block size is not too small.
- The global interpreter lock (GIL) is released while ctypes wrapped function is executing, allowing concurrency.
- Bifrost uses ctypesgen to generate wrappers from the underlying C++ code.

Example block: Fast Dispersion Measure Transform

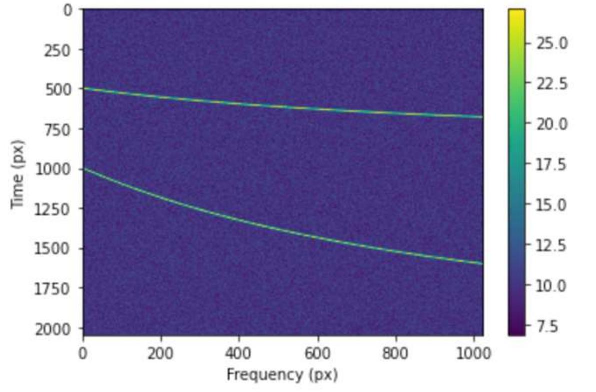
```
from bifrost.fdmt import Fdmt
fdmt = Fdmt()
```

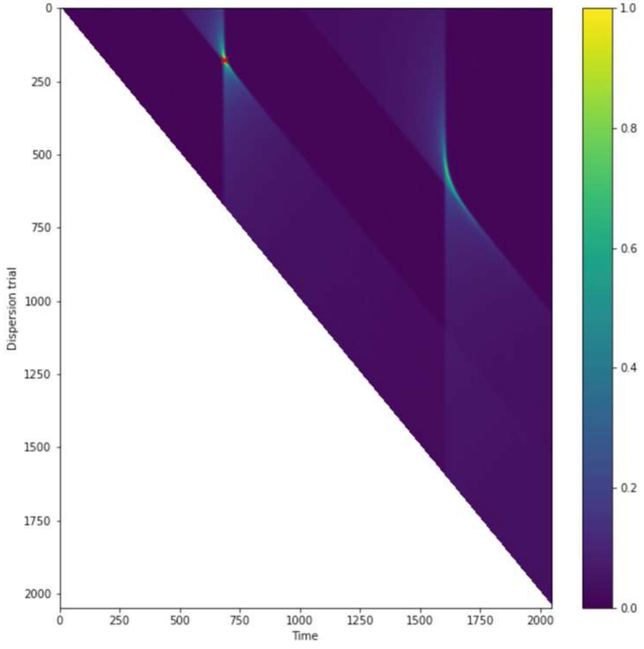
Initialize FDMT

```
[1]: max_delay = frame.data.shape[0]

     n_disp = max_delay
     n_time = frame.data.shape[0]
     n_chan = frame.data.shape[1]

     fdmt.init(n_chan, n_disp, frame.fch1 / 1e6, frame.df / 1e6)
```

```
# Input shape is (1, n_freq, n_time)
d_in = bf.ndarray(d_cpu, dtype='f32', space='cuda')
d_out = bf.ndarray(np.zeros(shape=(1, n_disp, n_time)), dtype='f32', space='cuda')

print(d_in.shape, d_out.shape)
print(n_chan, n_time, n_disp, n_time)

fdmt.execute(d_in, d_out, negative_delays=True)

d_out = d_out.copy(space='system')
```

```
frame.plot()
```





Bifrost map and ndarrays

A key under-the-hood part of `bifrost` is the `map` function, that provides a simple way to do fast arithmetic operations on the GPU. To get acquainted, we will ignore all of the pipeline infrastructure that bifrost provides, and just call the map function directly.

Let's suppose you have two `ndarrays` and wish to add them together on the GPU. Here is how you would do that with `map`:

```python
import bifrost as bf

# Create two arrays on the GPU, A and B, and an empty output C
a = bf.ndarray([1,2,3,4,5], space='cuda')
b = bf.ndarray([1,0,1,0,1], space='cuda')
c = bf.ndarray(np.zeros(5), space='cuda')

# Add them together
bf.map("c = a + b", data={'c': c, 'a': a, 'b': b})
print c
# ndarray([ 2.,  2.,  4.,  4.,  6.])
```

Recently added: cupy compatibility (2021)

bf map example kernel: applying beam weights

```
bf.map(self.kernel, {'a': idata, 'b': odata, 'w': w})
```

```
INT_CMULT_KERNEL = """
// Compute b = w * a

Complex<float> a_cf;
a_cf.real = a.real;
a_cf.imag = a.imag;

Complex<float> w_cf;
w_cf.real = w.real;
w_cf.imag = w.imag;

Complex<float> b_cf;
b_cf = a_cf * w_cf;

b.real = (int) b_cf.real;
b.imag = (int) b_cf.imag;
"""
```

High-level pipeline interface

UTMOST-2D fanbeam beamformer (for FRB search)

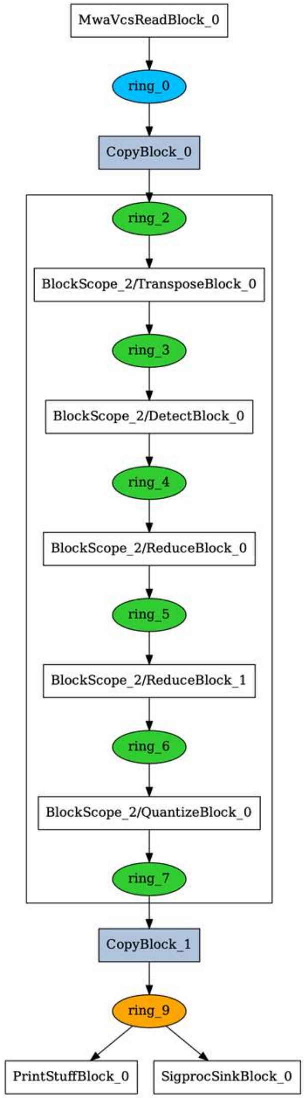
```
# GPU processing
b_gpu = bf.blocks.copy(b_dada, space='cuda', core=1, gpu=0)
with bf.block_scope(fuse=False, gpu=0):
    b_gpu = bf.views.merge_axes(b_gpu, 'station', 'pol')
    b_gpu = bf.blocks.transpose(b_gpu, ['time', 'subband', 'freq', 'heap', 'frame', 'snap', 'station'] )
    b_gpu = bf.views.merge_axes(b_gpu, 'subband', 'freq', label='freq')
    b_gpu = bf.views.merge_axes(b_gpu, 'snap', 'station', label='station')
    b_gpu = bf.views.merge_axes(b_gpu, 'heap', 'frame', label='fine_time')
    b_gpu = bf.views.add_axis(b_gpu, axis=3, label='pol')
    # Beanfarmer requires (time, freq, fine_time, pol, station)
    b_gpu = apply_weights(b_gpu, weights_callback=weightgen, output_dtype='ci8',
                          update_frequency=128)
    b_gpu = bf.blocks.beanfarmer(b_gpu, n_avg=n_avg, n_beam=n_beam, n_pol=n_pol,
                                 n_chan=n_chan, n_ant=n_ant, weights_file='beam_weights.hkl')
    PrintStuffBlock(b_gpu)

# Back to CPU and to disk
b_cpu = bf.blocks.copy(b_gpu, space='cuda_host', core=2)
#PrintStuffBlock(b_cpu)
h5write(b_cpu, outdir=outdir, prefix=file_prefix, n_int_per_file=n_int_per_file, core=3)
```

UTMOST-2D tied-array beamformer (pulsar timing)

```python
# First DADA buffer
if args.filename is None:
    b_dada = bf.blocks.psrdada.read_psrdada_buffer(args.buffer, hdr_callback, 1, core=0)
else:
    b_dada = bf.blocks.read_dada_file(args.filename.split(','), hdr_callback, gulp_nframe=1, core=0)
PrintStuffBlock(b_dada)

# GPU processing
b_gpu = bf.blocks.copy(b_dada, space='cuda', core=1, gpu=0)
with bf.block_scope(fuse=False, gpu=0):
    b_gpu = bf.views.merge_axes(b_gpu, 'station', 'pol')
    b_gpu = bf.blocks.transpose(b_gpu, ['time', 'subband', 'freq', 'heap', 'frame', 'snap', 'station'] )
    b_gpu = bf.views.merge_axes(b_gpu, 'subband', 'freq', label='freq')
    b_gpu = bf.views.merge_axes(b_gpu, 'snap', 'station', label='station')
    b_gpu = bf.views.merge_axes(b_gpu, 'heap', 'frame', label='fine_time')
    #b_gpu = bf.views.add_axis(b_gpu, axis=3, label='pol')
    # Beanfarmer requires (time, freq, fine_time, pol, station)
    b_gpu = apply_weights(b_gpu, weights_callback=wg, output_dtype='cf32',
                          update_frequency=weight_update_frequency)
    # The following only works if the input data are arranged X, Y, X, Y (as they are)
    # Remove to go back to the old approach that didn't handle polarisation
    b_gpu = bf.views.split_axis(b_gpu, 'station', 2, label='pol')
    b_gpu = bf.blocks.reduce(b_gpu, axis='station')

    if args.filterbank:
        #b_gpu = byip.detect(b_gpu, mode='stokes', axis='pol')
        b_gpu = byip.detect(b_gpu, mode='coherence', axis='pol')
        ### THIS IS THE OLD APPROACH that doesn't handle polarisation
        ##b_gpu = byip.detect(b_gpu, 'stokes_I')
        ##b_gpu = bf.views.add_axis(b_gpu, axis=3, label='pol')
        # Beanfarmer requires (time, freq, fine_time, pol, station)
        b_gpu = bf.blocks.reduce(b_gpu, 'fine_time', n_tavg)
    else:
        b_gpu = bf.blocks.quantize(b_gpu, 'ci8')

# Back to CPU and to disk
b_cpu = bf.blocks.copy(b_gpu, space='cuda_host', core=2)
PrintStuffBlock(b_cpu)
if args.outbuffer is not None:
    dada_key_hex = int(args.outbuffer, 16)
    b_dada = bf.blocks.psrdada.write_psrdada_buffer(b_cpu, dada_key_hex, gulp_nframe=1)
elif not args.benchmark:
    if args.filterbank: # Convert to filterbank
        b_cpu = bf.blocks.transpose(b_cpu, ['time', 'fine_time', 'station', 'pol', 'freq'])
        b_cpu0 = bf.views.merge_axes(b_cpu, 'time', 'fine_time')
        b_cpu0 = ExtractAntennaBlock2(b_cpu0, ant_id=0, nchan=args.nchan)
        bf.blocks.write_sigproc(b_cpu0, path=outdir)
    else: # Just write out voltages in h5 format
        h5write(b_cpu, outdir=outdir, prefix=file_prefix, n_int_per_file=n_int_per_file, core=3)
else:
    print("Enabling benchmark mode...")

print("Running pipeline")
bf.get_default_pipeline().shutdown_on_signals()
bf.get_default_pipeline().run()
```

A simple correlator for VCS data

```python
"""
# vcs_pipeline.py

A pipeline to convert MWA VCS data into filterbank files.
"""
import bifrost as bf
from blocks.read_vcs_sub import read_vcs_block
from blocks.print_stuff import print_stuff_block
from blocks.h5write import h5write_block

from logger import setup_logger
import time

setup_logger(filter="blocks.read_vcs", level="INFO")

if __name__ == "__main__":
    t0 = time.time()
    fn = '../fast-imaging-test/vcs/1164110416_metafits.fits'
    filelist = [fn, ]

    # Data arrive as ['time', 'coarse_channel',  'frame', 'fine_channel', 'station', 'pol']
    b_vcs    = read_vcs_block(filelist, space='cuda_host', gulp_nframe=1)
    b_gpu    = bf.blocks.copy(b_vcs, space='cuda')

    with bf.block_scope(fuse=True, gpu=0):
        b_gpu = bf.blocks.transpose(b_gpu, ['time', 'coarse_channel', 'fine_channel', 'station', 'pol', 'frame'])
        b_gpu = bf.views.merge_axes(b_gpu, 'coarse_channel', 'fine_channel', label='freq')
        b_gpu = bf.views.merge_axes(b_gpu, 'station', 'pol', label='station')
        b_gpu = bf.views.rename_axis(b_gpu, 'frame', 'fine_time')
        b_gpu = bf.blocks.correlate_dp4a(b_gpu, nframe_to_avg=10)
    b_cpu = bf.blocks.copy(b_gpu, space='system')
    print_stuff_block(b_cpu, n_gulp_per_print=10)
    h5write_block(b_cpu, prefix='correlator_test', n_int_per_file=10)

    print("Running pipeline")
    pipeline = bf.get_default_pipeline()
    pipeline.shutdown_on_signals()
    pipeline.dot_graph().render('vcs_pipeline_graph.log')
    pipeline.run()
```



Monitoring tools

6.2. Pipeline in /dev/shm

Details about the currently running bifrost pipeline are available in the `/dev/shm` directory. They are mapped into a directory structure (use the linux `tree` utility to view it):

```
dancpr@bldcpr:/bldata/bifrost/tools$ tree /dev/shm/bifrost
/dev/shm/bifrost
└── 17263
    └── Pipeline_0
        ├── AccumulateBlock_0
        │   ├── bind
        │   ├── in
        │   ├── out
        │   ├── perf
        │   └── sequence0
        ├── BlockScope_1
        │   ├── PrintHeaderBlock_0
        │   │   ├── bind
        │   │   ├── in
        │   │   ├── out
        │   │   ├── perf
        │   │   └── sequence0
        │   └── TransposeBlock_0
        │       ├── bind
        │       ├── in
        │       ├── out
        │       ├── perf
        │       └── sequence0
        ├── BlockScope_13
        ├...
```

6.3. like_top.py

The main performance monitoring tools is `like_top.py`. This is, as the name suggests, like the linux utility `top`.

..code:

```
like_top.py - bldcpr - load average: 0.59, 0.14, 0.05
Processes: 516 total, 1 running
CPU(s):  1.9%us,  1.4%sy,  0.0%ni, 84.5%id, 12.1%wa,  0.0%hi,  0.0%si,
Mem:   32341840k total,  19834116k used,  12507724k free,    515556k bu
Swap:  32938492k total,    767408k used,  32171084k free,  17982316k ca

  PID            Block  Core   %CPU    Total  Acquire  Process  Reserv
19154  GuppiRawSourceB     0    9.4    0.714    0.000    0.714    0.00
19154      FftBlock_0      3    4.4    0.733    0.699    0.034    0.00
19154     CopyBlock_0      2    4.4    0.722    0.700    0.021    0.00
19154  TransposeBlock_     1    3.5    0.710    0.695    0.015    0.00
19154  HdfWriteBlock_0     6    0.4    3.220    3.213    0.007    0.00
19154    DetectBlock_0     4    1.0    0.738    0.733    0.005    0.00
19154  FftShiftBlock_0     3    4.4    0.738    0.734    0.005    0.00
19154     CopyBlock_1      6    0.4    2.816    2.813    0.003    0.00
19154  AccumulateBlock     5    4.0    0.005    0.005    0.001    0.00
19154  PrintHeaderBloc    -1           3.220    3.220    0.000    0.00
```

- Acquire is the time spent waiting for input (i.e., waiting on upstream blocks),
- Process is the time spent processing data, and
- Reserve is the time spent waiting for output space to become available in the ring (i.e., waiting for downstream blocks).

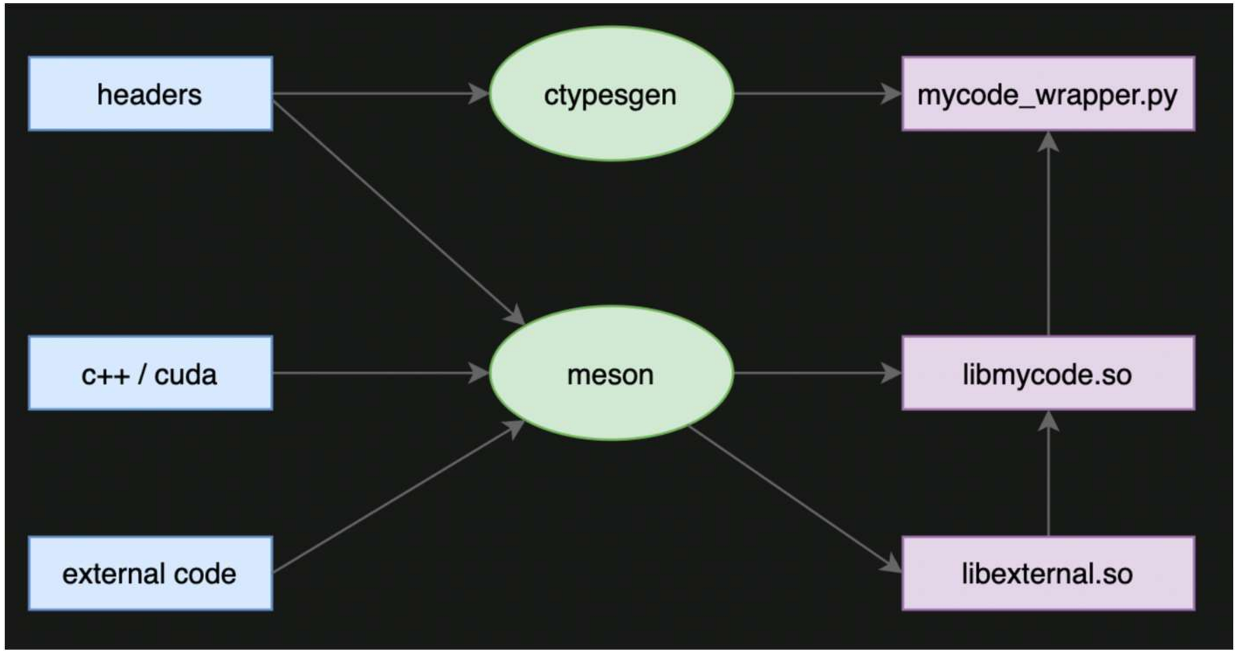
Note: The CPU fraction will probably be 100% on any GPU block because it's currently set to spin (busy loop) while waiting for the GPU.

Recent developments: plugin system

BRAGI RECEIVING A REALLY NICE APPLE FROM IDUNN (J. PENROSE, 1890)

- Bragr: A plugin system for bifrost to make it easy to incorporate third-party code. https://github.com/bf-plugins/bragr
- Generates a template using cookiecutter templating and meson build system.

Recent developments



- Bragr: A plugin system for bifrost to make it easy to incorporate third-party code. https://github.com/bf-plugins/bragr
- Generates a template using **cookiecutter** templating and **meson** build system.

More Recent Developments

Recent

- Autotools build system for core bifrost codebase.
  - Support for MacOS and C++17/17/20 features as needed.
- CUDA managed memory support.
- Support for complex 32-bit integers
- cupy interoperability.

Upcoming

- updated UDP capture using ibverbs.
- Inter-server RDMA transfer.
- Support for disk-based rings.

Proposed

- Porting CUDA to HIP (ADACS MAP program)

More Recent Developments

Recent

- Autotools build system for core bifrost codebase.
  - Support for MacOS and C++17/17/20 features as needed.
- CUDA managed memory support.
- Support for complex 32-bit integers
- cupy interoperability.

Upcoming

- updated UDP capture using ibverbs.
- Inter-server RDMA transfer.
- Support for disk-based rings.

Proposed

- Porting CUDA to HIP (ADACS MAP program)

Major credit to Jayce Dowell + Christopher League for new functionality

Continued work funded through NSF CSSI grant at UNM

Thank you

Jayce Dowell, Christopher League, Ben Barsdell, Danny Price, Miles Cranmer, Lincoln Greenhill, Greg Taylor + others