katgpucbf

Release 0.1.dev290+gf385556

SARAO DSP team

May 13, 2024

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 MeerKAT and MeerKAT Extension

The South African Radio Astronomy Observatory (SARAO) manages all radio astronomy initiatives and facilities in South Africa, including the MeerKAT radio telescope. MeerKAT is a precursor to the Square Kilometre Array (SKA) and consists of 64 offset-Gregorian antennas in the Karoo desert in South Africa.

MeerKAT Extension is a project currently underway to extend MeerKAT with additional antennas and longer baselines. This module (`katgpucbf`) is intended for deployment with MeerKAT Extension.

1.2 Radio Astronomy Correlators

Radio astronomy interferometry is a means of achieving higher sensitivity and resolution in radio telescopes by combining signals from multiple antennas, as opposed to relying on ever-larger individual antennas which are expensive and unwieldy.

This combination of signals is typically achieved digitally by equipment known as a correlator. In the narrowband case, correlation is achieved by cross- multiplying each antenna's signal with each other antenna's signal. Since modern radio telescopes have wideband receivers, the signal is decomposed (typically using the Fourier transform) into multiple narrowband frequency channels.

Mathematically, correlation and frequency decomposition can be done in any order. A correlator which calculates cross-correlations first and then frequency decomposition is referred to as X-F, while vice-versa is known as F-X. For practical reasons (both in terms of compute and interconnect), F-X correlators are more cost-effective to implement, and so MeerKAT and MeerKAT Extension make use of this architecture for their correlators.

MeerKAT and MeerKAT Extension's correlators were developed with the influence of CASPER. Costs are minimised by using commercially-available products wherever possible. In particular, Ethernet is used to connect signal-processing nodes. This eliminates the need to design costly custom backplanes.

1.3 This module

This module (`katgpucbf`) provides a software implementation of the DSP engines of a radio astronomy correlator described above.

The module contains several executable entry-points. The main functionality is implemented in **`fgpu`** and **`xbgpu`** which execute (respectively) an F- or an XB-engine. The F-engine implements the channelisation component of the correlator's operation, while the XB-engine calculates the correlation products (X) and beamformer output (B). The beamformer component is not currently implemented, but is planned for a future release.

Additionally, packet simulators are provided for testing purposes. A *Digitiser Packet Simulator* can be used to test either an F-engine or an entire correlator. An *F-Engine Packet Simulator* can be used to test an XB-engine in isolation.

The module also includes unit tests (`test/`), as well as a framework for automated testing of an entire correlator against the set of requirements applicable to the MeerKAT Extension CBF (`qualification/`).

As far as possible, the code in this package is not MeerKAT-specific. It could in theory be used at other facilities, provided that compatible input and output formats are used (including number of input and output bits). The *katgpucbf.meerkat* module contains some tables that are specific to MeerKAT and MeerKAT Extension, which are used by some convenience scripts, but which are not used by the core programs.

Some additional scripts (`scratch/`) which the developers have found to be useful are included, but user discretion is advised as these aren't subject to very much quality control, and will need to be adapted to your environment.

1.4 Controller

Todo: NGC-680 - Relationship with katsdpcontroller - reference to a later section which will describe it more thoroughly.

CHAPTER

TWO

MATHEMATICAL BACKGROUND

This section is not intended as a tutorial for radio astronomy. It briefly summarises some key equations, with the intent of illustrating implementation choices made by this package.

2.1 Frequencies

This package works only in positive baseband frequencies, and is unaware of heterodyne systems. Where necessary, the signal will need to be mixed to this range before being provided as input. As an example, MeerKAT L-band digitisers receive signal in the range 856–1712 MHz, but mix it down to 0–856 MHz by negating every second sample (the digital equivalent of a 856 MHz mixing signal).

This has implications for *delay compensation*.

2.2 Complex voltages

A wave with frequency f and wave number k is considered to have a phasor of

$$e^{(2\pi ft - kz)j}$$

where t is time and z is position. In particular, phase measured at a fixed position (an antenna) increases with time.

2.3 Polyphase filter bank

A finite impulse response (FIR) filter is applied to the signal to condition the frequency-domain response. The filter is the product of a Hann window (to reduce spectral leakage) and a sinc (to broaden the peak to cover the frequency bin). Specifically, if there are n output channels and t taps in the polyphase filter bank, then the filter has length $w = 2nt$, with coefficients

$$x_i = A \sin^2\left(\frac{\pi i}{w-1}\right) \operatorname{sinc}\left(w_c \cdot \frac{i + \frac{1}{2} - nt}{2n}\right),$$

where i runs from 0 to $w-1$. Here A is a normalisation factor which is chosen such that $\sum_i x_i^2 = 1$. This ensures that given white Gaussian noise as input, the expected output power in a channel is the same as the expected input power in a digitised sample. Note that the input and output are treated as integers rather than as fixed-point values.

The tuning parameter w_c (specified by the `--w-cutoff` command-line option) scales the width of the response in the frequency domain. The default value is 1, which makes the width of the response (at -6dB) approximately equal the channel spacing.

2.4 Correlation products

Given a baseline (p, q) and time-varying channelised voltages e_p and e_q, the correlation product is the sum of $e_p\overline{e_q}$ over the accumulation period. This is computed in integer arithmetic and so is lossless except when saturation occurs.

2.5 Narrowband

Todo: Document the down-conversion filter

2.6 Delay and phase compensation

The delay sign convention is such that a input voltage sample with timestamp t will have an output timestamp of $t + d$ (where d is the delay). In other words, the specified values are amounts by which the incoming signals should be delayed to align them.

To correctly apply delay with sub-sample precision, it is necessary to know the original ("sky") frequency of the signal, before mixing and aliasing to baseband. The user supplies this indirectly by specifying the phase correction that must be applied at the centre frequency, i.e. $-2\pi f_c d$, where f_c is the centre frequency. This calculation is provided by `katpoint.delay.DelayCorrection`.

CHAPTER

THREE

CHANGELOG

Version 0.1

- Initial release. Wideband correlator functionality is considered stable.

CHAPTER

FOUR

SYSTEM REQUIREMENTS

For basic operation (such as for development or proof-of-concept) the only hardware requirement is an NVIDIA GPU with tensor cores. The rest of this section describes recommended setup for high-performance operation.

4.1 Networking

An NVIDIA NIC (ConnectX or Bluefield) should be used, as katgpucbf can bypass the kernel networking stack when using one of these NICs. See the spead2 documentation for details on setting up and tuning the ibverbs support. Pay particular attention to disabling multicast loopback.

The correlator uses multicast packets to communicate between the individual engines. Your network needs to be set up to handle multicast, and to do so efficiently (i.e., not falling back to broadcasting). Note that the out-of-the-box configuration for Spectrum switches running Onyx allocates very little buffer space to multicast traffic, which can easily lead to lost packets. Refer to the manual for your switch to adjust the buffer allocations.

The engines also default to using large packets (8 KiB of payload, plus some headers), so your network needs to be configured to support jumbo frames. While there are command-line options to reduce the packet sizes, this will significantly reduce performance.

4.2 BIOS settings

See the system tuning guidance in the spead2 documentation. In particular, we've found that when running multiple F-engines per host on an AMD Epyc (Milan) system, we get best performance with

- NPS1 setting for NUMA per socket (NPS2 might work too, but NPS4 tends to cause sporadic lost packets);
- the GPU and the NIC in slots attached to different host bridges.

CHAPTER

FIVE

INSTALLATION

5.1 Installation with Docker

The recommended way to use katgpucbf is via Docker. There is currently no published Docker image, so it is necessary to build your own. To do so, change to the root directory of the repository and run

```
docker build -t NAME .
```

where *NAME* is the name to assign to the image.

You will need to have the NVIDIA container runtime installed to provide Docker with access to the GPU.

5.2 Installation with pip

It is also possible to install katgpucbf with pip. In this case, you will need to have CUDA already installed, as well as Vulkan drivers[1]. Change to the root directory of the repository and run

```
pip install ".[gpu]"
```

Note that if you are planning to do development on katgpucbf, you should refer to the *Developers' guide*.

[1] Standard driver installation should include Vulkan support, although you may need to install a package such as libvulkan1.

CHAPTER

SIX

OPERATION

There are two main scenarios involved in starting up and interacting with katgpucbf and its constituent engines:

1. the instantiation and running of a complete end-to-end correlator, and
2. the invocation of individual engines (dsim, fgpu, xbgpu) for more fine-grained testing and debugging.

The first requires a mechanism to orchestrate the simultaneous spin-up of a correlator's required components - that is, some combination of dsim(s), F-Engine(s) and XB-Engine(s). For this purpose, katgpucbf utilises the infrastructure provided by katsdpcontroller - discussed in the following section.

Regarding the testing and debugging of individual engines, more detailed explanations of their inner-workings are discussed in their respective, more dedicated-discussion documents.

The main thing to note is that, in both methods of invocation (via orchestration and individually), the engines support control via katcp commands issued to their `<host>:<port>`. `netcat` (nc) is likely the most readily-available tool for this job, but ntsh neatens up these exchanges and generally makes it easier to interact with.

6.1 katsdpcontroller

This package (katgpucbf) provides the components of a correlator (engines and simulators), but not the mechanisms to start up and orchestrate all the components as a cohesive unit. That is provided by katsdpcontroller.

For production use it is strongly recommended that katsdpcontroller is used to manage the correlator. Nevertheless, it is possible to run the individual pieces manually, or to implement an alternative controller. The remaining sections in this chapter describe the interfaces that are used by katsdpcontroller to communicate with the correlator components.

There are two parts to katsdpcontroller: a *master controller* and a *product controller*. There is a single product controller per instantiated correlator. It is responsible for:

- starting up the appropriate correlator components with suitable arguments, given a high-level description of the desired correlator configuration;
- monitoring the health of those components;
- registering them with Consul, so that infrastructure such as Prometheus can discover them;
- proxying their *katcp sensors*, so that clients need only subscribe to sensors from the product controller rather than individual components;
- in some cases, aggregating or renaming those sensors, to present a correlator-wide suite of sensors, without clients needing to know about the individual engines;
- providing additional correlator-wide katcp sensors;
- providing correlator-wide katcp requests, which are implemented by issuing similar but finer-grained requests to the individual engines.

The master controller manages product controllers (and hence correlators), starting them up and shutting them down on request from the user. In a system supporting subarrays, there will typically be a single master controller and zero or more product controllers at any one time.

It is worth noting that katsdpcontroller was originally written to control the MeerKAT Science Data Processor and later extended to control correlators, so it has a number of features, requests and sensors that are not relevant to correlators.

6.2 Starting the correlator

The katgpucbf repository comes with a `scratch/` directory, under which you will find handy scripts for correlator and engine invocation. Granted, the layout and usage of these scripts is tailored to SARAO DSP's internal lab development environment (e.g. host and interface names) and don't necessarily go through the same reviewing rigour as the actual codebase. For these reasons, it is recommended that these scripts are used more as an example of how to run components of katgpucbf, rather than set-in-stone modi operandi.

6.2.1 End-to-end correlator startup

If you intend on starting up a correlator with **`sim_correlator.py`**, you will require a running master controller in accordance with *katsdpcontroller*. The script itself provides an array of options for you to start your correlator; running `./sim_correlator.py --help` gives a brief explanation of the arguments required. Below is an example of a full command to run a 4k, 4-antenna, L-band correlator:

```
./sim_correlator -a 4 -c 4096 -i 0.5
--adc-sample-rate 1712e6
--name my_test_correlator
--image-override katgpucbf:harbor.sdp.kat.ac.za/dpp/katgpucbf:latest
lab5.sdp.kat.ac.za
```

The execution of this command contacts the master controller to request a new correlator product to be configured. The master controller figures out how many of each respective engine is required based on these input parameters, and launches them accordingly across the pool of processing nodes available.

6.2.2 Individual engine startup

The arguments required for individual engine invocation can be seen by running one of `{dsim, fgpu, xbgpu} --help` in an appropriately-configured terminal environment. There are a few mandatory ones, and ultimately stitching the entire incantation together by hand can become tiresome. For this reason, the scripts under `scratch/{fgpu, xbgpu}` have been shipped with the module.

The scripts for standalone engine usage are prepopulated with typical configuration values for your convenience, and are usually named **`run-{dsim, fgpu, xbgpu}.sh`**. It is important to note that the F- and XB-Engines can run in a standalone manner, but will require some form of stimulus to truly exercise the engine. For example, `fgpu` requires a corresponding `dsim` to produce data for ingest. Similarly, `xbgpu` requires an appropriately-configured `fsim`. Basically, the engines will do nothing until explicitly asked to.

Todo: `NGC-730` Update scratch directory to have a single config sub-directory. Also add comments on the scripts themselves to make it easier to follow.

Note: Before considering which engine you intend on testing, note the number of GPUs available in the target processing node. The CUDA library acknowledges the presence of a `CUDA_VISIBLE_DEVICES` environment variable,

similar to that discussed by katsdpsigproc. You can simply `export CUDA_VISIBLE_DEVICES=0` in your terminal environment for the engine invocation to acknowledge your intention of using a particular GPU.

To test a 4k, 4-antenna XB-Engine processing L-band data, use the following commands in separate terminals on two separate servers. This will launch a single *F-Engine Packet Simulator* on `host1` and a single **xbgpu** instance on `host2`:

```
[Connect to host1 and activate the local virtual environment]
(katgpucbf) user@host1:~/katgpucbf$ spead2_net_raw fsim --interface <interface name> --
↪ibv \
                                    --array-size 4 --channels 4096 \
                                    --channels-per-substream 1024 \
                                    239.10.10.10+1:7148
.
.
.
[Connect to host2 and activate the local virtual environment]
(katgpucbf) user@host2:~/katgpucbf$ spead2_net_raw numactl -C 1 xbgpu \
                                    --src-affinity 0 --src-comp-vector 0 \
                                    --dst-affinity 1 --dst-comp-vector 1 \
                                    --src-interface <interface name> \
                                    --dst-interface <interface name> \
                                    --src-ibv --dst-ibv \
                                    --adc-sample-rate 1712e6 --array-size 4 \
                                    --channels 4096 \
                                    --channels-per-substream 1024 \
                                    --samples-between-spectra 8192 \
                                    --katcp-port 7150 \
                                    239.10.10.10:7148 239.10.11.10:7148
```

Naturally, it is up to the user to ensure command-line parameters are consistent across the components under test, e.g. using the same `--array-size` is for the data generated (in the **fsim**) and the **xbgpu** instance.

Note: ibverbs requires `CAP_NET_RAW` capability on Linux hosts. See spead2's discussion on ensuring this is configured correctly for your usage.

Pinning thread affinities

Todo: `NGC-730` Update `run-{dsim, fpgu, xbgpu}.sh` scripts to standardise over usage of either `numactl` or `taskset`.

spead2's performance tuning discussion outlines the need to set the affinity of all threads that aren't specifically pinned by `--{src, dst}-affinity`. This is often the main Python thread, but libraries like CUDA tend to spin up helper threads.

Testing without a high-speed data network

katgpucbf allows the user to develop, debug and test its engines without the use of a high-speed e.g. 100GbE data network. The omission of `--{src, dst}-ibv` command-line parameters avoids receiving data via the Infiniband Verbs API. This means that if you wish to e.g. capture engine data on a machine that doesn't support ibverbs, you could use *tcpdump(8)*.

Note: The data rates you intend to process are still limited by the NIC in your host machine. To truly take advantage of running engines without a high-speed data network, consider reducing the `--adc-sample-rate` by e.g. a factor of ten as this value greatly affects the engine's data transmission rate.

6.3 Controlling the correlator

The correlator components are controlled using katcp. A user can connect to the `<host>:<port>` and issue a `?help` to see the full range of commands available. The `<host>` and `<port>` values for individual engines are configurable at runtime, whereas the `<host>` and `<port>` values for the correlator's *product controller* are yielded by the master controller after startup. Standard katcp requests (such as querying and subscribing to sensors) are not covered here; only application-specific requests are listed. Sensors are described in *katcp sensors*.

6.3.1 dsim

?signals *spec* [*period*]
: Change the signals that are generated. The signal specification is described in *Signal specification*. The resulting signal will be periodic with a period of *period* samples. The given period must divide into the `--max-period` command-line argument, which is also the default period if none is specified.

  The dither that is applied is cached on startup, but is independent for the different streams. Repeating the same command thus gives the same results, provided any randomised terms (such as `wgn`) use fixed seeds.

  It returns an ADC timestamp, which indicates the next sample which is generated with the new signals. This is kept for backwards compatibility, but the same information can be found in the `steady-state-timestamp` sensor.

?time
: Return the current UNIX timestamp on the server running the dsim. This can be used to get an approximate idea of which data is in flight, without depending on the dsim host and the client having synchronised clocks.

6.3.2 fgpu

?gain *stream input* [*values...*]
: Set the complex gains. This has the same semantics as the equivalent katsdpcontroller command, but *input* must be 0 or 1 to select the input polarisation.

?gain-all *stream values...*
: Set the complex gains for both inputs. This has the same semantics as the equivalent katsdpcontroller command.

?delays *stream start-time values...*
: Set the delay polynomials. This has the same semantics as the equivalent katsdpcontroller command, but takes exactly two delay model specifications (for the two polarisations).

6.3.3 xbgpu

?capture-start, ?capture-stop
: Enable or disable transmission of output data. This does not affect transmission of descriptors, which cannot be disabled. In the initial state transmission is disabled, unless the `--tx-enabled` command-line option has been passed.

6.4 Shutting down the correlator

6.4.1 End-to-end correlator shutdown

A user can issue a `?product-deconfigure` command to the correlator's product controller by connecting to its `<host>:<port>`. This command triggers the stop procedure of all engines and dsims running in the target correlator. More specifically:

- the product controller instructs the orchestration software to stop the containers running the engines,
- which is received by the engines as a `SIGTERM`,
- finally triggering a `halt` in the engines for a graceful shutdown.

The shutdown procedures are broadly similar between the dsim, fgpu and xbgpu. Ultimately they all:

- finish calculations on data currently in their pipelines,
- stop the transmission of their SPEAD descriptors, and
- in the case of `fgpu` and `xbgpu`, stop their `spead2` receivers, which allows for a more natural ending of internal processing operations.

6.4.2 Individual engine shutdown

Once you've sufficiently tested, debugged and/or reached the desired level of confusion, there are two options for engine shutdown:

1. simply issue a `Ctrl + C` in the terminal window where the engine was invoked, or
2. connect to the engine's `<host>:<port>` and issue a `?halt`.

After either of these approaches are executed, the engine will shutdown cleanly and quietly according to their common *Shutdown procedures*. As the *F-Engine Packet Simulator* is a simple CLI utility, the **`fsim`** just requires a `Ctrl + C` to end operations - no `katcp` commands supported here.

CHAPTER

SEVEN

MONITORING

There are two production mechanisms for monitoring components of the correlator: katcp sensors and Prometheus metrics.

These have different strengths and weaknesses. Prometheus is highly scalable (so can handle a large number of metrics), supports metric labels, and has a rich ecosystem (such as Grafana for easily visualising metrics). On the other hand, katcp sensors can be more precisely timestamped and support arbitrary string values (often containing structured data) rather than just floating-point.

In general, we use katcp sensors for information that should be archived alongside the data to facilitate its interpretation, as well as sensors that are needed by other subsystems in the MeerKAT telescope. Prometheus metrics are used for detailed health and performance monitoring. However, this rule is not hard-and-fast, and some information is reported via sensors for historical reasons.

There is a third monitoring mechanism (event monitoring) intended for development purposes.

7.1 katcp sensors

The katcp sensors are self-documenting: issuing a `?sensor-list` request to any of the servers will return a list of the sensors with descriptions. We thus limit this section to sensors that need a more detailed explanation.

It should be noted that a large number of sensors describing static configuration (number of channels, accumulation length and so on) are provided by the product controller rather than this module.

`steady-state-timestamp`
: This sensor is provided by both `dsim` and `fgpu`. It can be used to synchronise katcp requests with the data. After issuing a katcp request that will alter the data stream (such as `?signals`, `?gain` or `?delay`), query the sensor. It will contain an ADC timestamp. Any data received with that timestamp or greater will be up to date with the effects of all prior requests.

  It should be noted that a `?delay` request with a future load time is considered to have taken effect when the delay model has been updated, even if that load time has not yet been reached.

  It should also be noted that the sensor value from a `dsim` does not take into account any delays applied by F engines. One should add the delay of the corresponding F engine (or an upper bound on it) to obtain a safe timestamp for post-F engine data streams.

`signals`, `period` and `dither-seed` (dsim)
: To reproduce the output of the dsim exactly, it is necessary to save all three of these sensors, and pass the `dither-seed` back to the `--dither-seed` command-line option and the other two to the `?signals` request (as well as keeping other command-line arguments the same).

7.2 Prometheus metrics

To enable Prometheus metrics for any of the services, pass `--prometheus-port` to specify the port number. The metrics are then made available at the `/metrics` HTTP endpoint on that port. Pointing a web browser at that endpoint will show the available metrics with their documentation.

7.3 Event monitoring

It is also possible to perform very detailed monitoring of events occurring within fgpu and xbgpu, particularly related to time spent waiting on queues. This is intended for debugging performance issues rather than production use, as it has much higher overhead than the other monitoring mechanisms. To activate it, pass `--monitor-log` with a filename to the process. It will write a file with a JSON record per line. The helper script in **`scratch/plot.py`** can be used to show a visualisation of the various queues over time. It's not recommended for more than a few seconds of data.

CHAPTER

EIGHT

DATA INTERFACES

Todo: If this section gets to be too large, it can probably also make its way into its own file.

8.1 SPEAD Protocol

The Streaming Protocol for Exchanging Astronomical Data (SPEAD) is a lightweight streaming protocol, primarily UDP-based, designed for components of a radio astronomy signal-chain to transmit data to each other over Ethernet links.

The SPEAD implementation used in *katgpucbf* is `spead2`. It is highly recommended that consumers of *katgpucbf* output data also make use of `spead2`. For those who cannot, this document serves as a brief summary of the SPEAD protocol in order to understand the output of each application within *katgpucbf*, which are further detailed elsewhere.

SPEAD transmits logical collections of data known as *heaps*. A heap consists of one or more UDP packets. A SPEAD transmitter will decompose a heap into packets and the receiver will collect all the packets and reassemble the heap.

8.2 Packet Format

A number of metadata fields are included within each packet, to facilitate heap reassembly. The SPEAD flavour used in *katgpucbf* is 64-48, which means that each metadata field is 64 bits wide, with the first bit indicating the address mode, the next 15 carrying the item ID and the remaining 48 carrying the value (in the case of immediate items).

Each packet contains the following metadata fields:

`header`
: Contains information about the flavour of SPEAD being used.

`heap counter/id`
: A unique identifier for each new heap.

`heap size`
: Size of the heap in bytes.

`heap offset`
: Address in bytes indicating the current packet's location within the heap.

`payload size`
: Number of bytes within the current packet payload.

Each SPEAD stream will have additional 64-bit fields specific to itself, referred to in SPEAD nomenclature as *immediate items*. Each packet transmitted will contain all the immediate items to assist third-party consumers that prefer to

work at the packet level (see spead2.send.Heap.repeat_pointers — note that this is not default spead2 behaviour, but it is always enabled in katgpucbf).

Most of the metadata remains constant for all packets in a heap. The heap offset changes across packets, in multiples of the packet size (which is configurable at runtime). This is used by the receiver to reassemble packets into a full heap.

The values contained in the immediate items may change from heap to heap, or they may be static, with the data payload being the only changing thing, depending on the nature of the stream.

8.3 F-Engine Data Format

8.3.1 Input

The F-engine receives dual-polarisation input from a digitiser (raw antenna) stream. In MeerKAT and MeerKAT Extension, each polarisation's raw digitiser data is distributed over eight contiguous multicast addresses, to facilitate load-balancing on the network, but the receiver is flexible enough to accept input from more or fewer multicast addresses.

The only immediate item in the digitiser's output heap used by the F-engine is the `timestamp`.

8.3.2 Output Packet Format

In addition to the fields described in SPEAD's *Packet Format* above, the F-Engine's have an output data format as follows - formally labelled elsewhere as **Channelised Voltage Data SPEAD packets**. These immediate items are specific to the F-Engine's output stream.

`timestamp`
: A number to be scaled by an appropriate scale factor, provided as a KATCP sensor, to get the number of Unix seconds since epoch of the first time sample used to generate data in the current SPEAD heap.

`feng_id`
: Uniquely identifies the F-engine source for the data. A sensor can be consulted to determine the mapping of F-engine to antenna antenna input. The X-engine uses this field to distinguish data received from multiple F-engines.

`frequency`
: Identifies the first channel in the band of frequencies in the SPEAD heap. Can be used to reconstruct the full spectrum. Although each packet may represent a different frequency, this value remains constant across a heap and represents only the first frequency channel in the range of channels within the heap. The X-engine does not strictly need this information.

`feng_raw item pointer`
: Channelised complex data from both polarisations of digitiser associated with F-engine. Real comes before imaginary and input 0 before input 1. A number of consecutive samples from each channel are in the same packet.

The F-engines in an array each transmit a subset of frequency channels to each X-engine, with each X-engine receiving from a single multicast group. F-engines therefore need to ensure that their heap IDs do not collide.

8.4 X-Engine Data Format

8.4.1 Input

The X-Engine receives antenna channelised data from the output of the F-engines, as discussed above. Each X-Engine receives data from each F-engine, but only from a subset of the channels.

8.4.2 Output Packet Format

In addition to the fields described in SPEAD's *Packet Format* above, the X-Engine's have an output data format as follows - formally labelled elsewhere as **Baseline Correlation Products**. These immediate items are specific to the X-Engine's output stream.

`frequency`
: Identifies the first channel in the band of frequencies in the SPEAD heap. Although each packet represents a different frequency, this value remains constant across a heap and represents only the first frequency channel in the range of channels within the heap.

`timestamp`
: A number to be scaled by an appropriate scale factor, provided as a KATCP sensor, to get the number of Unix seconds since epoch of the first time sample used to generate data in the current SPEAD heap.

`xeng_raw item pointer`
: Integrated Baseline Correlation Products; packed in an order described by the KATCP sensor *xeng-stream-name*`-bls-ordering`. Real values are before imaginary. The bandwidth and centre frequencies of each sub-band are subject to the granularity offered by the X-engines.

In MeerKAT Extension, four correlation products are computed for each baseline, namely vv, hv, vh, and hh. Thus, for an 80-antenna correlator, there are $\frac{n(n+1)}{2} = 3240$ baselines, and 12960 correlation products. The parameter `n-bls` mentioned under `xeng_raw` refers to the latter figure.

Each X-engine sends data to its own multicast group. A receiver can combine data from several multicast groups to consume a wider spectrum, using the `frequency` item to place each heap. To facilitate this, X-engine output heap IDs are kept unique across all X-engines in an array.

CHAPTER

NINE

DSP ENGINE DESIGN

9.1 Terminology

We will use OpenCL terminology, as it is more generic. If you're more familiar with CUDA terminology, katsdpsigproc's introduction has a table mapping the most important concepts. For definitions of the concepts, refer to chapter 2 of the OpenCL specification. A summary of the most relevant concepts can also be found here.

9.2 Glossary

This section serves (hopefully) to clarify some potentially confusing terms used within the source code.

Chunk
: An array of data and associated metadata, including a timestamp. Chunks are the granularity at which data is managed within an engine (e.g., for transfer between CPU and GPU). To amortise per-chunk costs, chunks typically contain many SPEAD heaps.

Command Queue
: Channel for submitting work to a GPU. See `katsdpsigproc.abc.AbstractCommandQueue`.

Device
: GPU or other OpenCL accelerator device (which in general could even be the CPU). See `katsdpsigproc.abc.AbstractDevice`.

Engine
: A single process which consumes and/or produces SPEAD data, and is managed by katcp. An F-engine processes data for one antenna; an XB-Engine processes data for a configurable subset of the correlator's bandwidth. It is expected that a correlator will run more than one engine per server.

Event
: Used for synchronisation between command queues or between a command queue and the host. See `katsdpsigproc.abc.AbstractEvent`.

Heap
: Basic message unit of SPEAD. Heaps may comprise one or more packets.

Queue
: See `asyncio.Queue`. Not to be confused with Command Queues.

Queue Item
: See *`QueueItem`*. These are passed around on Queues.

Stream
: A stream of SPEAD data. The scope is somewhat flexible, depending on the viewpoint, and might span one or many multicast groups. For example, one F-engine sends to many XB-engines (using many multicast groups),

and this is referred to as a single stream in the fgpu code. Conversely, an XB-engine receives data from many F-engines (but using only one multicast group), and that is also called "a stream" within the xbgpu code.

This should not be confused with a CUDA stream, which corresponds to a Command Queue in OpenCL terminology.

Stream Group
: A group of incoming streams whose data are combined in chunks (see `spead2.recv.ChunkStreamRingGroup`). Stream groups can be logically treated like a single stream, but allow receiving to be scaled across multiple CPU cores (with one member `stream` per thread).

Timestamp
: Timestamps are expressed in units of ADC (analogue-to-digital converter) samples, measured from a configurable "sync time". When a timestamp is associated with a collection of data, it generally reflects the timestamp of the *first* ADC sample that forms part of that data.

9.3 Operation

The general operation of the DSP engines is illustrated in the diagram below: The F-engine uses two input streams and

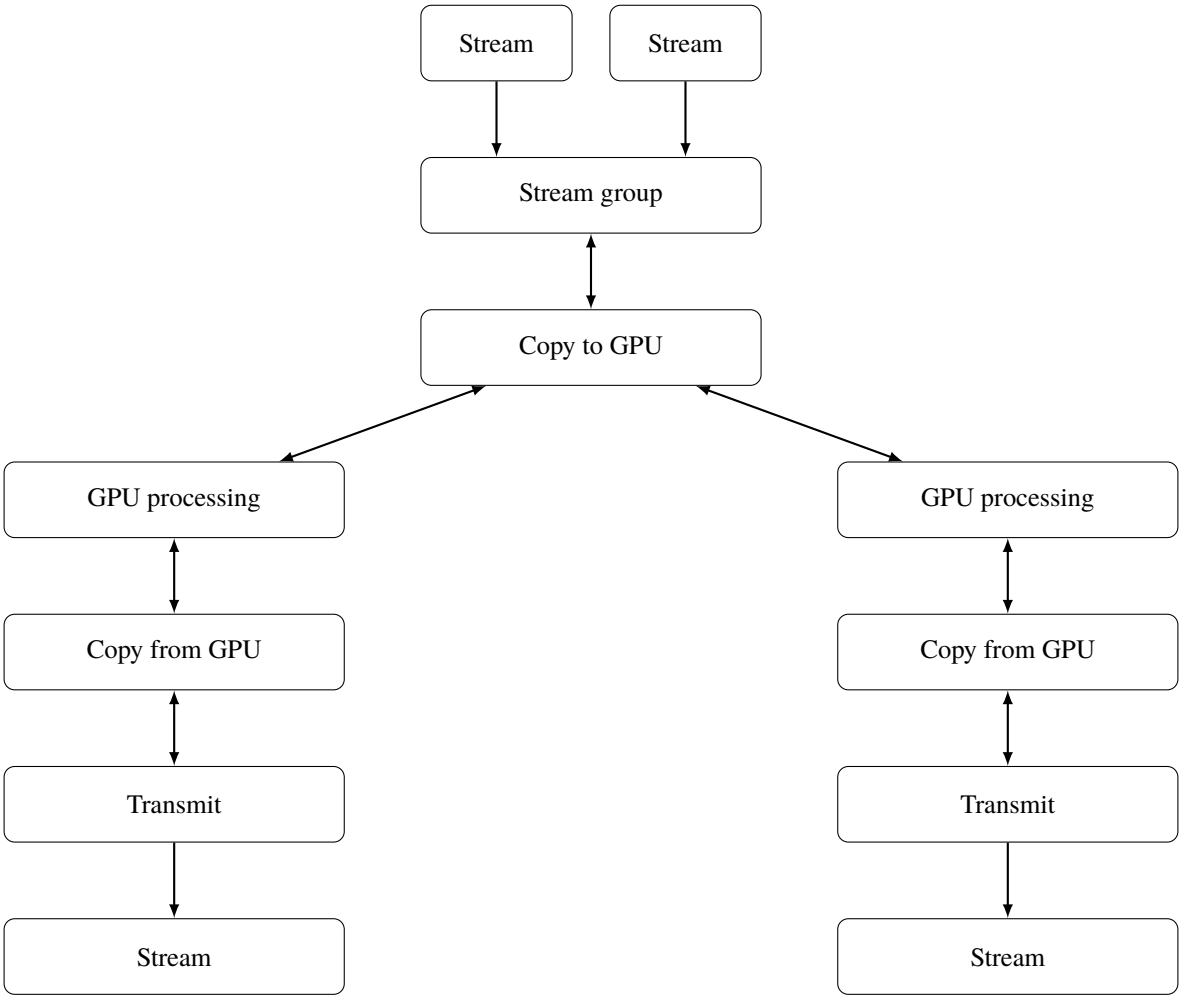


Fig. 1: Data Flow. Double-headed arrows represent data passed through a queue and returned via a free queue.

aligns two incoming polarisations, but in the XB-engine there is only one.

There might not always be multiple processing pipelines. When they exist, they are to support multiple outputs generated from the same input, such as wide- and narrow-band F-engines, or correlation products and beams. Separate outputs use separate output streams so that they can interleave their outputs while transmitting at different rates. They share a thread to reduce the number of cores required.

9.3.1 Chunking

GPUs have massive parallelism, and to exploit them fully requires large batch sizes (millions of elements). To accommodate this, the input packets are grouped into "chunks" of fixed sizes. There is a tradeoff in the chunk size: large chunks use more memory, add more latency to the system, and reduce LLC (last-level cache) hit rates. Smaller chunks limit parallelism, and in the case of the F-engine, increase the overheads associated with overlapping PFB (polyphase filter bank) windows.

Chunking also helps reduce the impact of slow Python code. Digitiser output heaps consist of only a single packet, and while F-engine output heaps can span multiple packets, they are still rather small and involving Python on a per-heap basis would be far too slow. We use `spead2.recv.ChunkRingStream` or `spead2.recv.ChunkStreamRingGroup` to group heaps into chunks, which means Python code is only run per-chunk.

9.3.2 Queues

Both engines consist of several components which run independently of each other — either via threads (spead2's C++ code) or Python's asyncio framework. The general pattern is that adjacent components are connected by a pair of queues: one carrying full buffers of data forward, and one returning free buffers. This approach allows all memory to be allocated up front. Slow components thus cause back-pressure on up-stream components by not returning buffers through the free queue fast enough. The number of buffers needs to be large enough to smooth out jitter in processing times.

A special case is the split from the receiver into multiple processing pipelines. In this case each processing pipeline has an incoming queue with new data (and each buffer is placed in each of these queues), but a single queue for returning free buffers. Since a buffer can only be placed on the free queue once it has been processed by all the pipelines, a reference count is held with the buffer to track how many usages it has. This should not be confused with the Python interpreter's reference count, although the purpose is similar.

9.3.3 Transfers and events

To achieve the desired throughput it is necessary to overlap transfers to and from the GPU with its computations. Transfers are done using separate command queues, and an CUDA/OpenCL event (see *the glossary*) is associated with the completion of each transfer. Where possible, these events are passed to the device to be waited for, so that the CPU does not need to block. The CPU does need to wait for host-to-device transfers before putting the buffer onto the free queue, and for device-to-host transfers before transmitting results, but this is deferred as long as possible.

The above concepts are illustrated in the following figure:

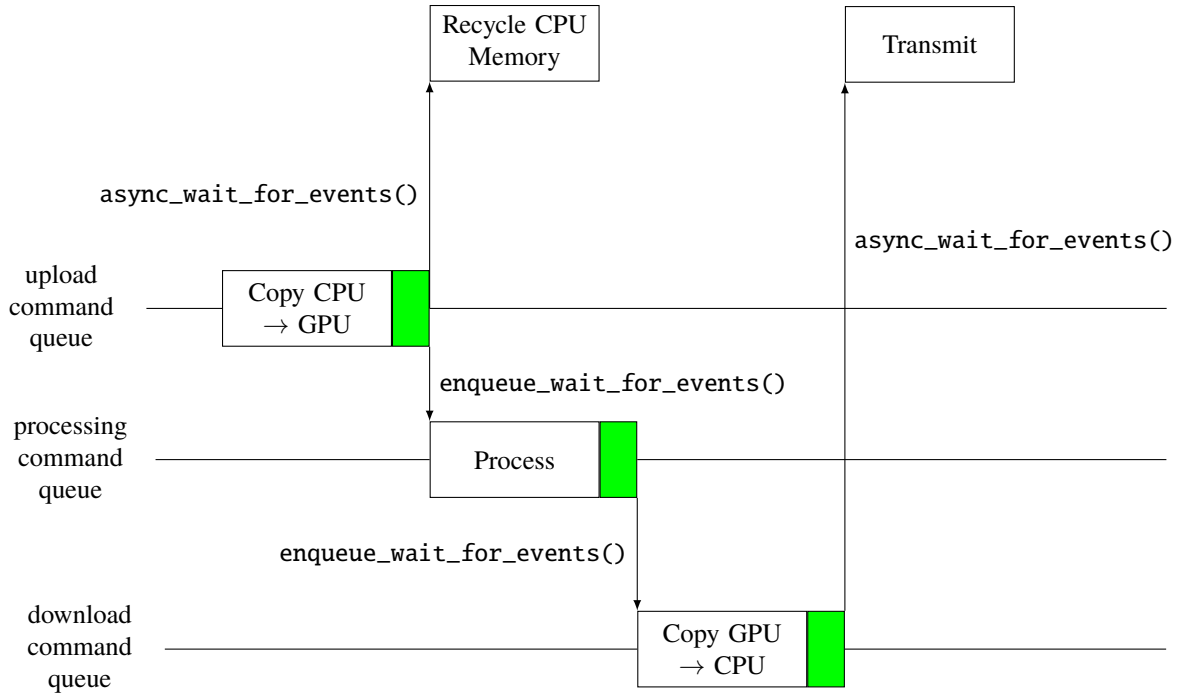


Fig. 2: GPU command queues, showing the upload, processing and download command queues, and the events (shown in green) used for synchronisation.

9.4 Common features

9.4.1 Shutdown procedures

The dsim, fgpu and xbgpu all make use of the `aiokatcp server`'s `on_stop` feature which allows for any engine-specific clean-up to take place before coming to a final halt.

The `on_stop` procedure is broadly similar between the dsim, fgpu and xbgpu.

- The `dsim` simply stops its internal calculation and sending processes of data and descriptors respectively.
- `fgpu` and `xbgpu` both stop their respective spead2 receivers, which allows for a more natural ending of internal processing operations.
  - Each stage of processing passes a *None*-type on to the next stage,
  - Eventually resulting in the engine sending a SPEAD stop heap across its output streams.

CHAPTER

TEN

F-ENGINE DESIGN

A somewhat higher-level view of the design can be found in [Merry2023]. The paper is based on an earlier version of this code (which pre-dates the narrowband support, for example).

We start by describing the design for wideband output. Narrowband outputs use most of the same components, but have extra complications, which are described in a *separate section*.

10.1 Network receive

The data from both polarisations are received by a single stream group, and polarisation forms the major axis of each chunk. The stream group may comprise multiple streams to support receive bandwidths that cannot be handled by a single thread. The polarisation of each packet is identified by a flag in the SPEAD items of the packet, rather than by the network interface or multicast group it was directed to.

To minimise the number of copies, chunks are initialised with CUDA pinned memory (host memory that can be efficiently copied to the GPU). Alternatively, it is possible to use vkgdr to have the CPU write directly to GPU memory while assembling the chunk. This is not enabled by default because it is not always possible to use more than 256 MiB of the GPU memory for this, which can severely limit the chunk size.

10.2 GPU Processing

The actual GPU kernels are reasonably straight-forward, because they're generally memory-bound rather than compute-bound. The main challenges are in data movement through the system.

10.2.1 Decode

Digitiser samples are bit-packed integers. While it is possible to write a dedicated kernel for decoding that makes efficient accesses to memory (using contiguous word-size loads), it is faster overall to do the decoding as part of the PFB filter because it avoids a round trip to memory. For the PFB, the decode is done in a very simple manner:

1. Determine the two bytes that hold the sample.
2. Load them and combine them into a 16-bit value.
3. Shift left to place the desired bits in the high bits.
4. Shift right to sign extend.
5. Convert to float.

While many bytes get loaded twice (because they hold bits from two samples), the cache is able to prevent this affecting DRAM bandwidth.

For the above to work, every sample needs to be entirely contained within two bytes. This will be the case for up to 10 bits, as well as for 12- or 16-bit samples, and hence these are the values supported. For 3-, 5-, 6- and 7-bit samples, the sample will sometimes but not always be contained in a single byte; in these cases an extraneous byte is loaded. This could be the byte following the end of the buffer; to handle this, a padding byte is added to avoid illegal memory accesses. For 2-, 4- and 8-bit samples, the value will always be contained in a single byte, and a simplified code path is used in these cases.

The narrowband digital down conversion also decodes the packed samples, but this is discussed *separately*.

10.2.2 Polyphase Filter Bank

The polyphase filter bank starts with a finite impulse response (FIR) filter, with some number of *taps* (e.g., 16), and a *step* size which is twice the number of output channels. This can be thought of as organising the samples as a 2D array, with *step* columns, and then applying a FIR down each column. Since the columns are independent, we map each column to a separate workitem, which keeps a sliding window of samples in its registers. GPUs generally don't allow indirect indexing of registers, so loop unrolling (by the number of taps) is used to ensure that the indices are known at compile time.

This might not give enough parallelism, particularly for small channel counts, so in fact each column in split into sections and a separate workitem is used for each section. There is a trade-off here as samples at the boundaries between sections need to be loaded by both workitems, leading to overheads.

Registers are used to hold both the sliding window and the weights, which leads to significant register pressure. This reduces occupancy and leads to reduced performance, but it is still good for up to 16 taps. For higher tap counts it would be necessary to redesign the kernel.

The weights are passed into the kernel as a table, rather than computed on the fly. While it may be possible to compute weights on the fly, using single precision in the computation would reduce the accuracy. Instead, we compute weights once on the host in double precision and then convert them to single precision.

A single FIR may also need to cross the boundary between chunks. To handle this, we allocate sufficient space at the end of each chunk for the PFB footprint, and copy the start of the next chunk to the end of the current one. Note that this adds an extra chunk worth of latency to the process.

10.2.3 FFT

After the FIR above, we can perform the FFT, which is done with cuFFT. The built-in support for doing multiple FFTs at once means that it can saturate the GPU even with small channel counts.

Naïvely using cuFFT for the full real-to-complex transformation can be quite slow and require multiple passes over the memory, because

1. There is a maximum number of channels that cuFFT can handle in one pass (it depends on the GPU, but seems to be 16384 for a GeForce RTX 3080 Ti). Larger transforms require at least one more pass.
2. It appears to handle real-to-complex transforms by first doing a complex-to-complex transform and then using an additional pass to fix up the result (i.e. form final FFT output).

For performance reasons, we move part of the Fourier Transform into the post-processing kernel, and also handle fixing up the real-to-complex transformation. This is achieved by decomposing the transformation into separately-computed smaller parts (using the Cooley-Tukey algorithm). Part of the Fourier transform is computed using cuFFT and the final stage (post-processing kernel) of the process includes one round of Cooley-Tukey computation and the computation to form the real-to-complex transformation.

To start, let's consider the traditional equation for the Fourier Transform. Let N be the number of channels into which we wish to decompose the input sequence, and let x_i be the (real) time-domain samples ($0 \leq i < 2N$) and X_k be its discrete Fourier transform (DFT). Because the time domain is real, the frequency domain is Hermitian symmetric, and we only need to compute half of it to recover all the information. We thus only need to consider k from 0 to $N - 1$ (this loses information about X_N, but it is convenient to discard it and thus have a power-of-two number of outputs).

$$X_k = \sum_{i=0}^{2N-1} e^{\frac{-2\pi j}{2N} \cdot ik} x_i.$$

We know that a direct implementation of the DFT is inefficient and alternative, more efficient means exist to perform this computation. One such method is the FFT introduced by Cooley-Tukey and in the GPU space cuFFT is one such implementation. As highlighted earlier, transform sizes of greater than 16384 (for a GeForce RTX 3080 Ti at least) require more than one memory pass making it less efficient than it needs to be. The technique detailed below uses the decomposition as provided by Cooley-Tukey to break down a larger transform into smaller 'sub-transforms' where the number of 'sub-transforms' is intentionally kept small for efficiency reasons and later combined (same process as the FFT) to form the larger transform size. This is a multi-step process and requires some extra notation and math tricks.

Real-to-complex transform

Now for some notation to see how this works. We start by treating x (a real array of length 2N) as if it is a complex array z of length N, with each adjacent pair of real values in x interpreted as the real and imaginary components of a complex value, and computing the Fourier transform of z. Formally, let $u_i = x_{2i}$ and $v_i = x_{2i+1}$. Then $z_i = u_i + jv_i = x_{2i} + jx_{2i+1}$.

We will start by computing the Fourier transform of z. Let U, V and Z denote the Fourier transforms of u, v and z respectively. Since the Fourier transform is a linear operator and we defined $z = u + jv$, we also have $Z = U + jV$.

It is important to remember that both u and v are real-valued, so U and V are Hermitian symmetric. By re-arranging things we can reconstruct U and V from Z using Hermitian symmetry properties. Let U' be U with reversed indices i.e., $U'_k = U_{-k}$ where indices are taken modulo N.

Hermitian symmetry means that $U'_k = U_{-k} = \overline{U_k}$ where the 'overline' in $\overline{U_k}$ denotes conjugation. This is effectively saying that by taking the reverse indices in U_k we get a conjugated result (see[1] for a reminder of why this is the case).

Looking back at U and V components, $U' = \overline{U}$ and similarly $V' = \overline{V}$. Why is this important? Previously we stated that $Z = U + jV$. Now we can consider the reverse of Z, namely Z'.

$$\begin{aligned} Z' &= U' + jV' \\ \overline{Z'} &= \overline{U' + jV'} \\ &= \overline{U'} + \overline{jV'} \\ &= U - jV \end{aligned}$$

What we actually want is to be able to separate out U and jV in terms of only Z and Z' (remember, z is the input array of real-valued samples reinterpreted as if it is an array of N complex samples).

Now let's formulate both U and V in terms of Z and $\overline{Z'}$.

$$\begin{aligned} Z + \overline{Z'} &= (U + jV) + (U - jV) \\ &= 2U + j(V - V) \\ &= 2U. \end{aligned}$$

[1] Going back to the original definition for the DFT we saw the complex exponential $e^{\frac{-2\pi j}{2N} \cdot ik}$ has a variable k where k represents the frequency component under computation for the input sequence x_i. If k is reversed (i.e. negative) the complex exponential changes to $e^{\frac{2\pi j}{2N} \cdot ik}$ as the negative in $-k$ multiplies out.

Likewise,

$$\begin{aligned} Z - \overline{Z'} &= (U + jV) - (U - jV) \\ &= 2jV. \end{aligned}$$

Using the above we can see that $U = \frac{Z+\overline{Z'}}{2}$ and similarly $V = \frac{Z-\overline{Z'}}{2j}$. Next, we use the Cooley-Tukey transform to construct X from U and V. To do this let's go back to the initial definition of the DFT and expand that using the Cooley-Tukey approach.

$$\begin{aligned} X_k &= \sum_{i=0}^{2N-1} e^{\frac{-2\pi j}{2N} \cdot ik} x_i \\ &= \sum_{i=0}^{N-1} e^{\frac{-2\pi j}{2N} \cdot 2ik} u_i + \sum_{i=0}^{N-1} e^{\frac{-2\pi j}{2N} \cdot (2i+1)k} v_i \\ &= \sum_{i=0}^{N-1} e^{\frac{-2\pi j}{N} \cdot ik} u_i + e^{\frac{-\pi j}{N} \cdot k} \sum_{i=0}^{N-1} e^{\frac{-2\pi j}{N} \cdot ik} v_i \\ &= U_k + e^{\frac{-\pi j}{N} \cdot k} V_k. \end{aligned}$$

What we get is a means to compute the desired output X_k using the U and V which we compute from the complex-valued input data sequence z.

We can also re-use some common expressions by computing X_{N-k} at the same time

$$\begin{aligned} X_{N-k} &= U_{N-k} + e^{\frac{-\pi j}{N} \cdot (N-k)} V_{N-k} \\ &= \overline{U_k} - \overline{e^{\frac{-\pi j}{N} \cdot k} V_k}. \end{aligned}$$

This raises the question: Why compute both X_k and X_{N-k}? After all, parameter k should range the full channel range initially stated (parameter N). The answer: compute efficiency. It is costly to compute U_k and V_k so if we can use them to compute two elements of X‘ (X_k and X_{N-k}) at once it is better than producing only one element of X.

Why is doing all this work more efficient that letting cuFFT handle the real-to-complex transformation? After all, cuFFT most likely does this (or something equivalent) internally. The answer is that instead of using a separate kernel for it (which would consume memory bandwidth), we built it into the postprocessing kernel (see the next section).

Unzipping the FFT

Right — lets get practical and show how we *actually* implement this. From here we'll assume all transforms are complex-to-complex unless specified otherwise. Firstly, some recap: the Cooley-Tukey algorithm allows a transform of size $N = mn$ to be decomposed into n transforms of size m followed by m transforms of size n. We'll refer to n as the "unzipping factor". We will keep it small (typically not more than 4), as the implementation requires registers proportional to this factor. We are now going to go step-by-step and separate the input array z into n parts of size m with each part operated on using a Fourier transform.

To recap the indexing used in the Cooley-Tukey algorithm: let a time-domain index i be written as $qn + r$ and a frequency-domain index k be written as $pm + s$. Let z^r denote the array $z_r, z_{n+r}, \ldots, z_{(m-1)n+r}$, and denote its Fourier transform by Z^r. It is worthwhile to point out that the superscript r *does not* denote exponentiation but rather is a means to indicate an r^{th} array. In practice this r^{th} array is a subset (part) of the larger z array of input data.

As a way of an example, let $n = 4$ ("unzipping factor") and $N = 32768$ (total number of channels). Now let's unpack this a bit further — what is actually happening is that the initial array z is divided into $n = 4$ separate arrays each of $m = 32768/4 = 8192$ elements (hence the $N = mn$ above). The actual samples that land up in each array are defined by the indices i and k.

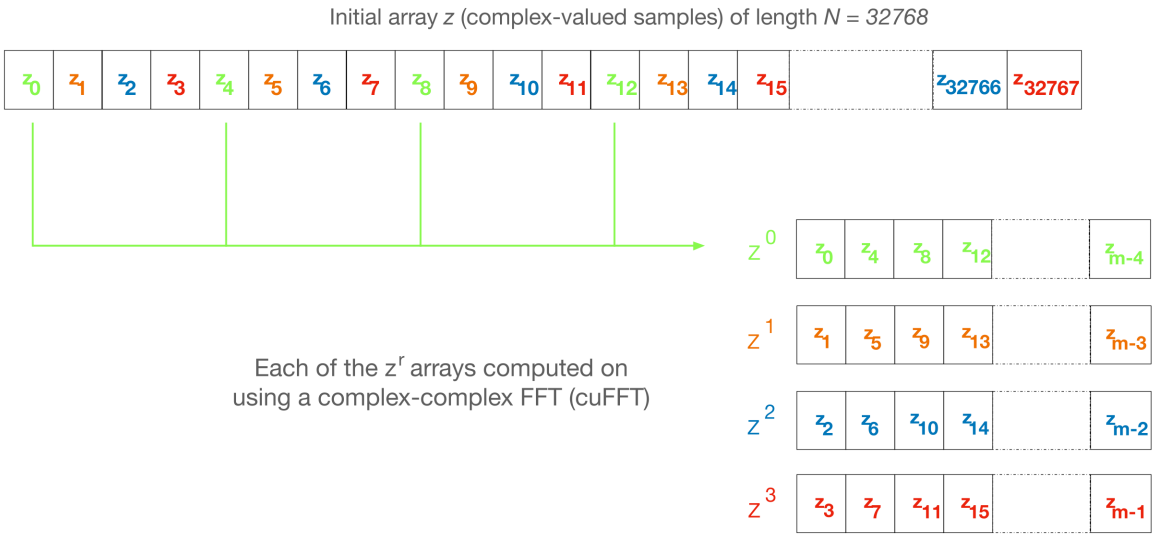
Lets start with i. It was stated that $i = qn + r$. The parameter r takes on the range 0 to $n - 1$ (so $r = 0$ to $r = 3$ as $n = 4$) and q takes on the range 0 to $m - 1$ (i.e. $q = 0$ to $q = 8191$). So we are dividing up array z into n smaller arrays denoted by r (i.e. z^r) each of length $m = 8192$. So what does this look like practically?

The first array when $r = 0$ (i.e. z^0)

Inputs qn + r	Index i
$0 \cdot 4 + 0$	0
$1 \cdot 4 + 0$	4
$2 \cdot 4 + 0$	8
...	...
...	...
$8191 \cdot 4 + 0$	32764

This can be extended to the other remaining arrays. The fourth array when $r = 3$ (for example), z^3 is z_3, z_7, z_{11}, ..., z_{32767}.

What this shows is that each sub-array consists of samples from the initial array z indexed by $i = qn + r$ where each sample is every 4^{th} and offset by r. Pictorially this looks like,



Right, so we have separate sub-arrays as indexed from the initial array, what happens next? These various z^r arrays are fed to cuFFT yielding n complex-to-complex transforms. These separate transforms now need to be combined to form a single real-to-complex transform of the full initial size. An inconvenience of this structure is that z^r is not a contiguous set of input samples, but a strided array. While cuFFT does support both strided inputs and batched transformations, we cannot batch over r and over multiple spectra at the same time as it only supports a single batch dimension with corresponding stride. We solve this by modifying the PFB kernel to reorder its output such that each z^r is output contiguously. This can be done by shuffling some bits in the output index (because we assume powers of two everywhere).

To see how the k indexing works $k = pm + s$ and is dealt with in a similar manner as above. Parameter $m = 8192$ (in this example), and p has a range 0 to $n - 1$ (i.e. $p = 0$ to $p = 3$ as $n = 4$ in our example); and s takes on the range 0 to $m - 1$ (i.e. $s = 0$ to $s = 8191$).

Looking at this practically,

When $p = 0$

Inputs pm + s	Index k
$0 \cdot 8192 + 0$	0
$0 \cdot 8192 + 1$	1
$0 \cdot 8192 + 2$	2
...	...
...	...
$0 \cdot 8192 + 8191$	8191

This too can be extended to the other remaining arrays.

Viewing the above tables it can be seen that the full range of outputs are indexed in batches of $m = 8192$ outputs, *but*, this is not yet the final output and are merely the outputs as provided by inputting the respective z^r arrays into cuFFT (all we have done at this point is computed Z^r using cuFFT). As a useful flashback, we are aiming to compute Z_k from z (made up from smaller arrays z^r) with the intention of computing the U and V terms. Why? So that with U and V we can compute X_k which is our desired final output.

The aim is to compute Z_k so putting it more formally we have

$$\begin{aligned}
Z_k = Z_{pm+s} &= \sum_{i=0}^{mn-1} e^{\frac{-2\pi j}{mn} \cdot ik} z_i \\
&= \sum_{q=0}^{m-1} \sum_{r=0}^{n-1} e^{\frac{-2\pi j}{mn}(qn+r)(pm+s)} z_{qn+r} \\
&= \sum_{r=0}^{n-1} e^{\frac{-2\pi j}{n} \cdot rp} \left[e^{\frac{-2\pi j}{mn} \cdot rs} \sum_{q=0}^{m-1} e^{\frac{-2\pi j}{m} \cdot qs} z_q^r \right] \\
&= \sum_{r=0}^{n-1} e^{\frac{-2\pi j}{n} \cdot rp} \left[e^{\frac{-2\pi j}{mn} \cdot rs} Z_s^r \right].
\end{aligned}$$

The whole expression is a Fourier transform of the expression in brackets (the exponential inside the bracket is the so-called "twiddle factor").

In the post-processing kernel, each work-item computes the results for a single s and for all p. To compute the real-to-complex transformation, it also needs to compute

$$\overline{Z_{-k}} = \overline{Z_{-pm-s}} = \sum_{r=0}^{n-1} e^{\frac{-2\pi j}{n} \cdot rp} \left[e^{\frac{-2\pi j}{mn} \cdot rs} \overline{Z_{-s}^r} \right].$$

Right, lets wrap things up. We have Z_k (i.e. Z) and $\overline{Z_{-k}}$ (i.e. $\overline{Z'}$) which is what we set out to compute. This then means we can compute X_k and X_{N-k} as stated earlier from $U = \frac{Z+\overline{Z'}}{2}$ and $V = \frac{Z-\overline{Z'}}{2j}$ (with appropriate twiddle factor) to combine the various outputs from cuFFT and get the final desired output X_k.

We also wish to keep a tally of saturated (clipped) values, which requires that each output value is considered exactly once. This is made more complicated by the process that computes X_k and X_{N-k} jointly. With $k = pm + s$, we consider all $0 \leq p < n$ and $0 \leq s \leq \frac{m}{2}$, and discard X_{N-k} when $s = 0$ or $s = \frac{m}{2}$ as these are duplicated cases.

10.2.4 Postprocessing

The remaining steps are to

1. Compute the real Fourier transform from several complex-to-complex transforms (see the previous section).
2. Apply gains and fine delays.
3. Do a partial transpose, so that *spectra-per-heap* spectra are stored contiguously for each channel (the Nyquist frequencies are also discarded at this point).
4. Convert to integer.
5. Where the output bits per sample is not a whole number of bytes, do the necessary bit-packing.
6. Interleave the polarisations.

These are all combined into a single kernel to minimise memory traffic. The katsdpsigproc package provides a template for transpositions, and the other operations are all straightforward. While C++ doesn't have a convert with saturation function, we can access the CUDA functionality through inline PTX assembly (OpenCL C has an equivalent function).

Fine delays and the twiddle factor for the Cooley-Tukey transformation are computed using the `sincospi` function, which saves both a multiplication by π and a range reduction.

The gains, fine delays and phases need to be made available to the kernel. We found that transferring them through the usual CUDA copy mechanism leads to sub-optimal scheduling, because these (small) transfers could end up queued behind the much larger transfers of digitiser samples. Instead, we use vkgdr to allow the CPU to write directly to the GPU buffers. The buffers are replicated per output item, so that it is possible for the CPU to be updating the values for one output item while the GPU is computing on another.

10.2.5 Coarse delays

One of the more challenging aspects of the processing design was the handling of delays. In the end we chose to exploit the fact that the expected delay rates are very small, typically leading to at most one coarse delay change per chunk. We thus break up each chunk into sections where the coarse delay is constant for both polarisations.

Our approach is based on inverting the delay model: output timestamps are regularly spaced, and for each output spectrum, determine the sample in the input that will be delayed until that time (to the nearest sample). We then take a contiguous range of input samples starting from that point to use in the PFB. Unlike the MeerKAT FPGA F-engine, this means that every output spectrum has a common delay for all samples. There will also likely be differences from the MeerKAT F-engine when there are large discontinuities in the delay model, as the inversion becomes ambiguous.

The polarisations are allowed to have independent delay models. To accommodate different coarse delays, the space at the end of each chunk (to which the start of the following chunk is copied to accommodate the PFB footprint) is expanded, to ensure that as long as one polarisation's input starts within the chunk proper, both can be serviced from the extended chunk. This involves a tradeoff where support for larger differential delays requires more memory and more bandwidth. The dominant terms of the delay are shared between polarisations, and the differential delay is expected to be extremely small (tens of nanoseconds), so this has minimal impact.

The GPU processing is split into a front-end and a back-end: the front-end consists of just the PFB FIR, while the backend consists of FFT and post-processing. Because changes in delay affect the ratio of input samples to output spectra, the front-end and back-end may run at different cadences. We run the front-end until we've generated enough spectra to fill a back-end buffer, then run the back-end and push the resulting spectra into a queue for transmission. It's important to (as far as possible) always run the back-end on the same amount of data, because cuFFT bakes the number of FFTs into its plan.

10.2.6 Digitiser sample statistics

The PFB kernel also computes the average power of the incoming signal. Ideally that would be done by a separate kernel that processed each incoming sample exactly once. However, doing so would be expensive in memory bandwidth. Instead, we update statistics as samples are loaded for PFB calculations.

Some care is needed to avoid double-counting due to overlapping PFB windows. The simplest way to add this to the existing code is that for each output spectrum, we include the last $2 \times$ channels samples from the PFB window. In steady state operation and in the absence of coarse delay changes, this will count each sample exactly once. Coarse delay changes will cause some samples to be counted twice or not at all, but these are sufficiently rare that it is not likely to affect the statistics.

Average power is updated at the granularity of output chunks. The PFB kernel updates a total power accumulator stored in the output item. This is performed using (64-bit) integer arithmetic, as this avoids the pitfalls of floating-point precision when accumulating a large number of samples.

10.3 Network transmit

The current transmit system is quite simple. By default a single spead2 stream is created, with one substream per multicast destination. For each output chunk, memory together with a set of heaps is created in advance. The heaps are carefully constructed so that they reference numpy arrays (including for the timestamps), rather than copying data into spead2. This allows heaps to be recycled for new data without having to create new heap objects.

If the traffic for a single engine exceeds the bandwidth of the network interface, it is necessary to distribute it over multiple interfaces. In this case, several spead2 streams are created (one per interface). Each of them has a substream for every multicast destination, but they are not all used (the duplication simplifies indexing). When heaps are transmitted, a stream is selected for each heap to balance the load. Descriptors and stop heaps are just sent through the first stream for simplicity. This scheme assumes that all the interfaces are connected to the same network and hence it does not matter which interface is used other than for load balancing.

10.3.1 PeerDirect

When GPUDirect RDMA / PeerDirect is used, the mechanism is altered slightly to eliminate the copy from the GPU to the host:

1. Chunks no longer own their memory. Instead, they use CUDA device pointers referencing the memory stored in an OutItem. As a result, Chunks and OutItems are tied tightly together (each OutItem holds a reference to the corresponding Chunk), instead of existing on separate queues.
2. Instead of OutItems being returned to the free queue once the data has been copied to the host, they are only returned after the data they hold has been fully transmitted.
3. More OutItems are allocated to compensate for the increased time required before an OutItem can be reused. This has not yet been tuned.

There may be opportunities for further optimisation, in the sense of reducing the amount of memory that is not actively in use, because some parts of an OutItem can be recycled sooner than others. Since GPUs that support this feature tend to have large amounts of memory, this is not seen as a priority.

10.3.2 Output Heap Payload Composition

In the case of an 8192-channel array with 64 X-engines, each heap contains 8192/64 = 128 channels. By default, there are 256 time samples per channel. Each sample is dual-pol complex 8-bit data for a combined sample width of 32 bits or 4 bytes.

The heap payload size in this example is equal to

channels_per_heap * samples_per_channel * complex_sample_size = 128 * 256 * 4 = 131,072 = 128 KiB.

The payload size defaults to a power of 2, so that packet boundaries in a heap align with channel boundaries. This isn't important for the `spead2` receiver used in the X-engine, but it may be useful for potential third party consumers of F-engine data.

10.4 Missing data handling

Inevitably some input data will be lost and this needs to be handled. The approach taken is that any output heap which is affected by data loss is instead not transmitted. All the processing prior to transmission happens as normal, just using bogus data (typically whatever was in the chunk from the previous time it was used), as this is simpler than trying to make vectorised code skip over the missing data.

To track the missing data, a series of "present" boolean arrays passes down the pipeline alongside the data. The first such array is populated by spead2. From there a number of transformations occur:

1. When copying the head of one chain to append it to the tail of the previous one, the same is done with the presence flags.
2. A prefix sum (see `numpy.cumsum()`) is computed over the flags of the chunk. This allows the number of good packets in any interval to be computed quickly.
3. For each output spectrum, the corresponding interval of input heaps is computed (per polarisation) to determine whether any are missing, to produce per-spectrum presence flags.
4. When an output chunk is ready to be sent, the per-spectrum flags are reduced to per-frame flags.

10.5 Narrowband

Narrowband outputs are those in which only a portion of the digitised bandwidth is channelised and output. Typically they have narrower channel widths. The overall approach is as follows:

1. The signal is multiplied (*mixed*) by a complex tone of the form $e^{2\pi jft}$, to effect a shift in the frequency of the signal. The centre of the desired band is placed at the DC frequency.
2. The signal is convolved with a low-pass filter. This suppresses most of the unwanted parts of the band, to the extent possible with a FIR filter.
3. The signal is subsampled (every Nth sample is retained), reducing the data rate. The low-pass filter above limits aliasing. At this stage, twice as much bandwidth as desired is retained.
4. The rest of the pipeline proceeds largely as before, but using double the final channel count (since the bandwidth is also doubled, the channel width is as desired). The input is now complex rather than real, so the Fourier transform is a complex-to-complex rather than real-to-complex transform.
5. Half the channels (the outer half) are discarded.

Note: To avoid confusion, the "subsampling factor" is the ratio of original to retained samples in the subsampling step, while the "decimation factor" is the factor by which the bandwidth is reduced. Because the mixing turns a real

signal into a complex signal, the subsampling factor is twice the decimation factor in step 3 (but equal to the overall decimation factor).

The decimation is thus achieved by a combination of time-domain (steps 2 and 3) and frequency domain (step 5) techniques. This has better computational efficiency than a purely frequency-domain approach (which would require the PFB to be run on the full bandwidth), while mitigating many of the filter design problems inherent in a purely time-domain approach (the roll-off of the FIR filter can be hidden in the discarded outer channels).

The first three steps are implemented by a "digital down-conversion" ("DDC") kernel. This is applied to each input chunk, after copying the head of the following chunk to the tail of the chunk. This does lead to redundant down-conversion in the overlap region, and could potentially be optimised.

The PFB FIR kernel has alternations because it needs to consume single-precision complex inputs rather than packed integers. However, the real and imaginary components are independent, and so the input is treated internally as if it contained just real values, with an adjustment to correctly index the weights. The postprocessing kernel also has adjustments, as the corrections for a real-to-complex Fourier transform are no longer required, and the outer channels must be discarded.

An incidental difference between the wideband and narrowband modes is that in wideband, the DC frequency of the Fourier transform corresponds to the lowest on-sky frequency, while for wideband it corresponds to the centre on-sky frequency. This difference is also handled in the postprocessing kernel. Internally, channels are numbered according to the Fourier transform (0 being the DC channel), but different calculations are used in wideband versus narrowband mode to swap the two halves of the band (and to discard half the channels) when

- indexing the gains array;
- indexing the output array;
- computing the phase from the fine delay and channel.

10.5.1 Down-conversion kernel

For efficiency, the first three operations above are implemented in the same kernel. In particular, the filtered samples that would be removed by subsampling are never actually computed. Unlike the memory-bound PFB kernel, this kernel is at the boundary between being memory-bound and compute-bound (depending on the number of taps). The design thus needs a more efficient approach to decoding the packed samples.

Each work-item is responsible for completely computing C consecutive output values (C is a tuning parameter), which it does concurrently. We can describe the operation with the equation

$$y_{b+i} = \sum_{k=0}^{T-1} x_{S(b+i)+k} \cdot h_k \cdot e^{2\pi j f [S(b+i)+k]}$$

where

- x is the input
- y is the output
- h contains the weights
- S is the subsampling factor
- T is the number of taps
- b is the first of the C outputs to produce
- $0 \leq i < C$ is the index into the C outputs to produce; and
- f is the frequency of the mixer signal, in cycles per digitiser sample.

The first simplification we make is to pre-compute the weights with the mixer (on the CPU). Let $z_k = h_k e^{2\pi j f k}$. Then the equation becomes

$$y_{b+i} = e^{2\pi j f S(b+i)} \sum_{k=0}^{T-1} x_{S(b+i)+k} \cdot z_k.$$

For now we'll focus on just the summation, and deal with the exponential later. For simplicity, assume T is a multiple of S (although the implementation does not require it) and let $W = \frac{T}{S}$. Then we can write k as $pS + q$ and rewrite this equation as

$$y_{b+i} = e^{2\pi j f S(b+i)} \sum_{p=0}^{W-1} \sum_{q=0}^{S-1} x_{S(b+i+p)+q} \cdot z_{pS+q}.$$

The kernel iterates first over q, then p, then i. A separate accumulator variable is kept for each value of i. For a given q, we only need $C + W - 1$ different values of x (since that's the range of $i + p$). We decode them all into an array before iterating over q and i to update the accumulators.

When q is advanced, we need to decode a new set of $C + W - 1$ samples. These immediately follow the previous set. We take advantage of this: in many cases, the new sample occupies (at least partially) the same 32-bit word from which we obtained the previous sample. By keeping those $C + W - 1$ words around, we get a head-start on decoding the new sample.

Choosing C is a trade-off. Larger values of C clearly increase register pressure. However they reduce the number of loads required: each work item decodes $(C + W - 1)S$ samples to produce C outputs, which is an average of $S + \frac{(W-1)S}{C}$. To further reduce the global memory traffic, all the samples and weights are copied to local memory at the start of the kernel. The results are also first transposed in local memory before being written back to global memory, to improve the global memory access pattern.

The implementation relies heavily on loop unrolling. Provided that CS samples occupy a whole number of 32-bit words (so that different work-items are loading samples from the same bit positions within a word), all the conditional logic involved in decoding the samples can be evaluated at compile-time.

Mixer signal

Care needs to be taken with the precision of the argument to the mixer signal. Simply evaluating the sine and cosine of $2\pi ft$ when t is large can lead to a catastrophic loss of precision, as the product ft will have a large integer part and leave few bits for the fractional part. Even passing f in single precision can lead to large errors.

To avoid these problems, fixed-point computations are used. Phase is represented as a fractional number of cycles, scaled by 2^{32} and stored in a 32-bit integer. When performing arithmetic on values encoded this way, the values may overflow and wrap. The high bits that are lost represent complete cycles, and so have no effect on phase.

10.5.2 Filter design

Discarding half the channels after channelisation allows for a lot of freedom in the design of the DDC FIR filter: the discarded channels, as well as their aliases, can have an arbitrary response. This allows for a gradual transition from passband to stopband. We use `scipy.signal.remez()` to produce a filter that is as close as possible to 1 in the passband and 0 in the stopband. A weighting factor (which the user can override) balances the priority of the passband (ripple) and stopband (alias suppression).

The filter performance is slightly improved by noting that the discarded channels have multiple aliases, and the filter response in those aliases is also irrelevant. We thus use `scipy.signal.remez()` to only optimise the response to those channels that alias into the output.

10.5.3 Delays

Coarse delay is (as for wideband) implemented using an input offset to the PFB FIR kernel. This means that the resolution of coarse delay is coarser than for wideband (by the subsampling factor). This choice is driven by the access patterns in the various kernels: the DDC kernel depends on knowing at compile time where each packed sample starts within a word, and hence is not amenable to single-sample input offsets.

10.5.4 Multiple outputs

A standard use case for MeerKAT is to produce wideband and narrowband outputs from a single input stream. To make this efficient, a single engine can support multiple output streams, and the input is only transferred to the GPU once.

The code is split into an Engine class that handles common input tasks, and a Pipeline class that handles per-output processing and transmission. Copying the head of each chunk to the tail of the previous chunk is handled by the Engine, after which the previous chunk is pushed to the input queue of each Pipeline. The chunks have reference counts to help determine when all pipelines are done with them.

10.5.5 Input statistics

The wideband PFB FIR kernel also computes statistics on the input digitiser stream (just RMS power, at the time of writing). Since all the outputs are produced from the same input, we do not attempt to duplicate this calculation for narrowband.

An engine with only narrowband outputs will thus be lacking these statistics. Calculating the statistics in that case would require extending the DDC kernel to compute the same statistics.

CHAPTER

ELEVEN

XB-ENGINE DESIGN

11.1 Correlation

11.1.1 Implementation details of correlation kernel

The correlation kernel of the XB-engine is implemented using a modified version of code originally written by John Romein of ASTRON (which can be accessed here).

For an overview of the tensor-core correlator compute kernel, see the GTC presentation. This section gets into the low-level details of the implementation. Familiarity with CUDA (including warp matrix multiplies) as well as the function of a correlator are assumed.

Complex multiplications

The tensor cores can only perform calculations on real numbers. The correlation kernel takes complex numbers as adjacent pairs of real and imaginary parts, and constructs matrices in a way that allows complex multiplications to be performed. The GTC presentation shows diagrams of this, so only a brief explanation will be given.

Consider a correlation product $A \times B^H$. The real part of $(a_r + ja_i)(\overline{b_r + jb_i})$ is $a_r b_r + a_i b_i$, so simply by placing the real and imaginary parts in alternate columns of A and B, each dot product between rows will give the desired real part.

The imaginary part is $a_i b_r - a_r b_i$, so we could construct a second matrix B' containing $(-b_i, b_r)$ in each pair of adjacent real entries (instead of (b_r, b_i)), and then the product would give the imaginary part. If instead of using two separate matrices, we interleave the rows of B and B', the resulting product will interleave the original products (by column) and hence place the real and imaginary components together again.

In the code this transformation is implemented by `conj_perm()`. It uses some bit manipulations to perform the calculation on a 32-bit integer that potentially has multiple samples. This code assumes a little-endian architecture, which is all that CUDA supports. Let's consider the code for `NR_BITS` of 8:

```
__byte_perm(v, 0x00FF00FF - (v & 0xFF00FF00), 0x2705);
```

In this case `v` contains two complex numbers. The mask selects the imaginary components. These are effectively subtracted from 0 to negate them; the left-hand side of the subtraction is `0x00FF00FF` rather than 0 so that the lower component doesn't cause a carry that affects the higher component. The other two bytes of the result are not relevant. Next, `__byte_perm()` selects the four desired bytes in the appropriate order.

It should be noted that this transformation will map -128 to itself rather than +128, because +128 cannot be represented in a two's-complement int8. The caller is responsible for ensuring that this input value is not used.

Parameters and constants

NR_BITS specifies the type of the incoming samples:

- 4 means each byte contains a packed 4-bit real and 4-bit imaginary.
- 8 means the real and imaginary components are signed bytes.
- 16 means half-precision float.

NR_SAMPLES_PER_CHANNEL is the number of samples in time processed in a single call to the kernel. These are divided into groups of NR_TIMES_PER_BLOCK, which is the number loaded into shared memory at a time before computing with them. Changing NR_TIMES_PER_BLOCK would require substantial changes to the loading code: the fetches are hard-coded to use a certain number of bits of the thread ID to index this dimension. Increasing it significantly (e.g., to match the 256 that is native to MeerKAT) would probably require too much shared memory.

NR_RECEIVERS_PER_BLOCK refers to the size of the subsets of antenna data in the input matrix which will be correlated per thread block. It has three possible values (32, 48 and 64) which correspond to processing 32×32, 48×48 or 64×32 (*not* 64×64) regions of the correlation matrix. The kernel uses NR_RECEIVERS_PER_BLOCK_X for the second dimension.

NR_CHANNELS is the number of channels over which to correlate, but there seems to be little need for this to be baked into the kernel. It only forms the outermost dimension of the inputs and outputs, and the Y axis of the thread grid, and could just as easily be dynamic.

NR_RECEIVERS_PER_TCM_X and NR_RECEIVERS_PER_TCM_Y are the number of (dual-pol) receivers per warp matrix multiply. Keeping in mind that the "Y" receiver corresponds to rows (and to aSamples temporary storage, with "X" corresponding to bSamples), this is 8×4 (8×2 for 4-bit samples). With dual-pol receivers that equates to 16×8 inputs. The reason it is not 16×16 (to match the matrix shape supported by the tensor cores) is the expansion of the B matrix for complex multiplication as described above.

In doCorrelateRectangle(), nrFragmentsX and nrFragmentsY indicate the number of "fragments" (tensor-core matrices) that the *warp* (not the thread block) is responsible along each dimension.

Thread indexing

There is a hard-coded value of 4 warps per block, arranged as 32×2×2. The first axis simply determines the position within a warp. The other two axes are used for different purposes in different parts of the code. Most typically, they subdivide the output block into quadrants (so for example a 64×32 output block is divided into four 32×16 output blocks, with one warp responsible for computing each). In loading code, the threadIdx is flattened into a 1D index (tid).

The thread grid is 2D. The y axis indicates the channel, while the x axis selects an output block within the output triangle. Some trickery with square roots is used to perform this mapping.

When NR_RECEIVERS_PER_BLOCK is 32 or 48, the output space is dealt with in square blocks, in doCorrelateRectangle(). The correlation matrix is conjugate symmetric, so this involves computing some redundant elements, which are discarded as part of storeVisibilities(). When it is 64, things get more complicated: certain blocks are processed with doCorrelateTriangle(), which is optimised for blocks that lie on the main diagonal. The figure above illustrates the arrangement for a 192-antenna array. The numbers in white boxes are the block IDs (blockIdx.x). Each green block is processed with doCorrelateRectangle(); it is shown divided into four quadrants (corresponding to the warps) and further subdivided into the fragments computed by each warp. The red/blue blocks are processed with doCorrelateTriangle(). The three blue regions are processed using warps 1-3 (a lookup table indicates the starting position), while the three red areas in each triangle are handled by warp 0.

When NR_BITS is 4 the situation is very similar, but the fragments are 8×2 instead of 8×4.

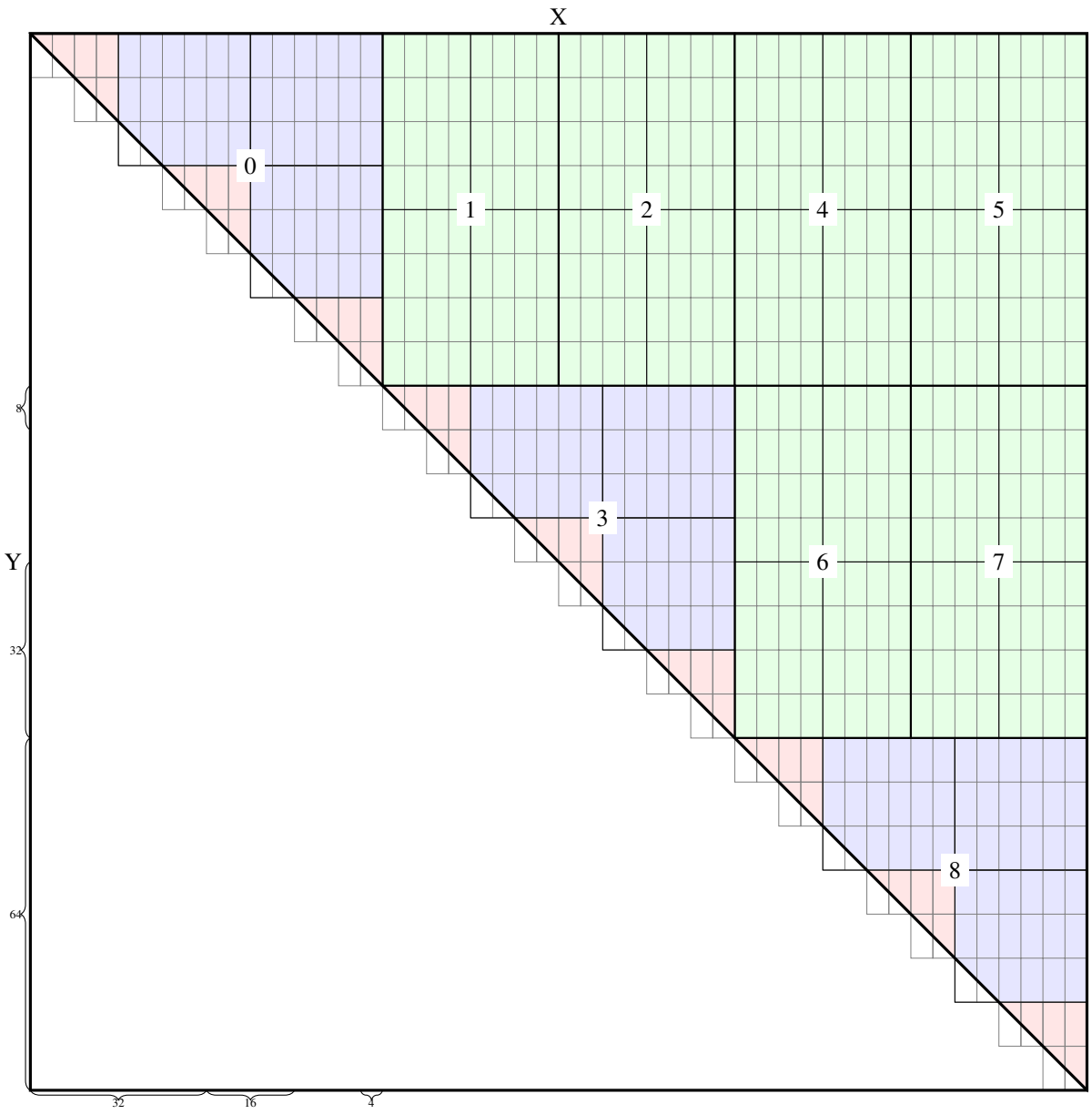


Fig. 1: Block, warp and fragment layout when `NR_RECEIVERS_PER_BLOCK` is 64 and `NR_BITS` is 8 or 16.

Data loading

A batch of voltage samples is loaded into shared memory, then used from there. Since each warp is computing multiple output fragments, each voltage is used by multiple matrix multiplies, and so caching them in shared memory reduces global memory traffic. The shared memory is also double-buffered, which is presumably to increase instruction-level parallelism and reduce the number of synchronisations required.

Rather than perform loads using the natural type of the samples, they are performed using wide types like `int4`, presumably to make more efficient use of the memory type, and type-casts pointers to access the raw memory. It should be noted that this sort of type-punning is undefined behaviour in C++, but there doesn't seem to be a safer alternative (`memcpy` is safe but it works one byte at a time, which destroyed performance).

Loading is implemented using the `FetchData` class. At construction time it takes thread-specific offsets to the receiver (antenna), polarisation and time. The `load()` member functions takes base channel, time and receiver that are uniform across the block. If the specific element to access is outside the bounds, the data is not loaded and left as zero.

Asynchronous loading

Note: The asynchronous loading support has been removed in the katgpucbf fork, as it was not really compatible with the axis reordering. This section is left as a reference should it be brought back in future.

When there is support in hardware (Compute Capability 8.0 or later, i.e., Ampere) and a new enough CUDA version, an asynchronous memory copy is used for extra latency hiding (or possibly to reduce register pressure). It's implemented using an experimental (and deprecated) version of the API; for reference one needs to read the 11.1 CUDA programming guide rather than the latest version.

The `READ_AHEAD` macro is slightly confusing. Let's assume a large enough `NR_SAMPLES_PER_CHANNEL` that `READ_AHEAD` is 2 and `NR_SHARED_BUFFERS` is 4. Then the following can all be occurring simultaneously:

1. Reading from shared buffer i to do the computations.
2. Asynchronous copies to shared buffers $i + 1$ to $i + 3$, inclusive (note that accounts for 3 buffers, not 2).

Within a single thread there can only be two async copies outstanding while doing the computations, because before starting computation on a buffer it waits for the copy targeting that buffer to complete. But because there is no call to `__syncthreads()` between the end of computation and the scheduling of the following copy, the scenario above can occur overall, with different threads in different parts of the loop. This explains why 4 buffers are needed.

Result storage

The result storage is particularly complicated in an attempt to optimise the process. CUDA says that the `fragment` type has implementation-defined memory layout, and the individual matrix elements can only be portably read by using `store_matrix_sync()` to write the results to shared or global memory. The memory layouts supported by this function don't correspond to the packed triangular shape the kernel wants, so some extra steps are required.

For a set of recognised architectures, the elements of the fragment class are read directly, using knowledge of the architecture-specific memory layout. In the fallback case (where `PORTABLE` is defined), the fragment is written to shared-memory scratch space then read back to extract the elements.

The upstream code is designed to do all the accumulation inside the kernel, by passing in all the data for the entire dump. While this is efficient (only writing results to global memory once), it would limit the dump period based on the available memory. In katgpucbf, the code has been modified so that results are added to the existing values in global memory.

11.1.2 Accumulations, Dumps and Output Data

The input data is accumulated before being output. For every output heap, multiple input heaps are received.

A heap from a single F-Engine consists of a set number of spectra, referred to as `spectra_per_heap`, where the spectra are time samples. Each of these time samples is part of a different spectrum, meaning that the timestamp difference per sample is equal to the value of `--samples-between-spectra`. The timestamp difference between two consecutive heaps from the same F-Engine is equal to:

*heap_timestamp_step = spectra_per_heap * samples_between_spectra.*

The value of `spectra_per_heap` is not set explicitly on the command line, but rather inferred from the `--jones-per-batch` argument. The latter is the product of `spectra_per_heap` and the stream's channel count (and thus, the number of Jones vectors in an F-engine output batch).

A *batch* of heaps is a collection of heaps from different F-Engines with the same timestamp. A *chunk* consists of multiple consecutive batches (the number is given by the option `--heaps-per-fengine-per-chunk`). Correlation generally occurs on a chunk at a time, accumulating results. The correlation kernel is modified in several ways to support this:

- The `FetchData` class splits the time index into a batch index and an offset within the batch, so that the rest of the code can be oblivious to batches, and just work in time "blocks" (`NR_TIMES_PER_BLOCK` spectra).
- The various functions take a runtime range of blocks to process.
- To provide more parallelism (important when each engine is processing only a few channels), the grid has an extra Z dimension. The time blocks to be processed are divided amongst the values of `blockIdx.z`. There is a trade-off here: one wants to serially accumulate over as many blocks as possible to reduce the final global memory traffic for writing back results. To avoid race conditions in accumulation, each value of `blockIdx.z` uses a separate global-memory accumulator.

An accumulation period is called an *accumulation* and the data output from that accumulation is normally called a *dump* — the terms are used interchangeably. Once all the data for a dump has been correlated, the separate accumulators are added together ("reduced") to produce a final result. This reduction process also converts from 64-bit to 32-bit integers, saturating if necessary, and counts the number of saturated visibilities.

The number of batches to accumulate in an accumulation is equal to the `--heap-accumulation-threshold` flag. The timestamp difference between successive dumps is therefore equal to:

*timestamp_difference = spectra_per_heap * samples_between_spectra * heap_accumulation_threshold*

The output heap timestamp is aligned to an integer multiple of *timestamp_difference* (equivalent to the current SKARAB "auto-resync" logic). The total accumulation time is equal to:

*accumulation_time_s = timestamp_difference * adc_sample_rate(Hz)* seconds.

The output heap contains multiple packets and these packets are distributed over the entire *accumulation_time_s* interval to reduce network burstiness. The default configuration in `katgpucbf.xbgpu.main` is for 0.5 second dumps when using the MeerKAT 1712 MSps L-band digitisers.

The dump boundaries are aligned to whole batches, but may fall in the middle of a chunk. In this case, each invocation of the correlation kernel will only process a subset of the batches in the chunk.

Output Heap Payload Composition

Each correlation product contains a real and imaginary sample (both 32-bit integer) for a combined size of 8 bytes per baseline. The ordering of the correlation products is given in the *xeng-stream-name*-bls-ordering sensor in the product controller, but can be calculated deterministically: `get_baseline_index()` indicates the ordering of the baselines, and the four individual correlation products are always ordered `aa, ba, ab, bb`, where *a* and *b* can either be vertical or horizontal polarisation (`v` or `h`), depending on the configuration of the instrument.

All the baselines for a single channel are grouped together contiguously in the heap, and each X-engine correlates a contiguous subset of the entire spectrum. For example, in an 80-antenna, 8192-channel array with 64 X-engines, each X-engine output heap contains 8192/64 = 128 channels.

The heap payload size in this example is equal to

> *channels_per_heap * correlation_products * complex_sample_size = 128 * 12960 * 8 = 13,271,040 bytes or 12.656 MiB.*

11.1.3 Missing Data Handling

As with fgpu, metadata indicating present or missing input heaps are passed down the pipeline alongside the data. If some input data is missing, processing is performed as normal. Unlike fgpu which suppresses transmissions for which some input data was missing, xbgpu will replace affected baselines with a "magic number" of (-2**31, 1), so that unaffected baselines can still be transmitted, but the receiver will know that those baselines are invalid. If a dump is affected by missing data on all antennas, it will still be transmitted but will contain only the magic value and no useful data.

11.2 Beamforming

11.2.1 GPU kernel

For the low number of beams required for MeerKAT (8 single-pol beams), the beamforming operation is entirely limited by the GPU DRAM bandwidth. There is thus no benefit to using tensor cores, and potentially significant downsides (implementation complexity and reduced accuracy for steering coefficients). The kernel thus uses standard single-precision floating-point operations, and is written to maximise bandwidth usage.

The beamforming operation consists of two phases: coefficient generation and coefficient application. To minimise use of system memory, the coefficients are generated on the fly in the same kernel, rather than computed in a separate kernel and communicated through global memory. This design allows for more dynamic coefficients in the future, such as delays that are updated much more frequently according to a formula.

The external interface to the beamformer has four parameters per beam-antenna pair: a weight, a quantisation gain (common to all antennas), a delay and a fringe-rate. All except the delay are combined by the CPU into a single (complex) weight per beam-antenna pair, and the delay is scaled into a convenient unit for computing the phase slope. The final coefficient applied to channel c, beam b, antenna a is

$$W_{abc} = w_{ab} e^{j\pi c d_{ab}}$$

where w_{ab} and d_{ab} are the weight and delay values passed to the kernel.

Each workgroup of the kernel handles multiple spectra and all beams and antennas, but only a single channel. Conceptually, the kernel first computes W_{abc} for all antennas and beams and stores it to local memory, then applies it to all antennas and beams. Each input sample is loaded once before it is used for all beams. An accumulator is maintained for each beam. Since each coefficient is used many times (the number depends on the work group size, which is a tuning parameter, but 64-256 is reasonable) after it is computed, the cost for computing coefficients is amortised.

In practice, this would cause local memory usage to scale without bound as the number of antennas increases. To keep it bounded (for a fixed number of beams), the antennas are processed in batches, computing then applying W_{abc} for each batch before starting the next batch. Larger batch sizes have two advantages:

1. The two phases in each batch need to be separated by a barrier to coordinate access to the shared memory. Larger batches reduce the number of barriers.
2. If the batch size is small, the number of coefficients to compute is also small, and there is not enough work to keep all the work items busy, making the coefficient computation less efficient.

Higher beam counts

The design above works well for small numbers of beams (up to about 64 single-pol beams), but the register usage scales with the number of beams and eventually the registers spill to memory, causing very poor performance.

To handle more beams, the kernel batches over beams, just as it does over antennas. The beam batch loop becomes an outer loop, with the rest of the kernel operating as before but only on a single batch.

This does mean that the inputs are loaded multiple times, but caches help significantly here, and the kernel tends to be be more compute-bound in this domain.

Dithering

To improve linearity, a random value in the interval (-0.5, 0.5) is added to each component (real and imaginary) before quantisation. These values are generated using curand, with its underlying XORWOW generator. It is designed for parallel use, with each thread having the same seed but a different *sequence* parameter to `curand_init()`. This minimises correlation between sequences generated by different threads. The sequence numbers are also chosen to be distinct between the different engines, to avoid correlation between channels.

Floating-point rounding issues make it tricky to get a perfectly zero-mean distribution. While it is probably inconsequential, simply using `curand_uniform(state) - 0.5f` will not give zero mean. We solve this by mapping the 2^{32} possible return values of `curand()` to the range $(-2^{31}, 2^{31})$ with zero represented twice, before scaling to convert to a real value in $(-0.5, 0.5)$. While this is still a deviation from uniformity, it does give a symmetric distribution.

CHAPTER

TWELVE

DEVELOPMENT ENVIRONMENT

12.1 Setting up a development environment

First, clone the repo from Github.

```
git clone git@github.com:ska-sa/katgpucbf.git
```

A setup script (**dev-setup.sh**) is included for your convenience to get going.

```
cd katgpucbf
source dev-setup.sh
```

The script will perform the following actions:

- Create a fresh Python virtual environment.
- Install all the requirements for running, developing and building this documentation.
- Install the *katgpucbf* package itself, in editable mode.
- Build this documentation.
- Install **pre-commit** to help with keeping things tidy.

Sourcing the script instead of executing it directly will keep your virtual environment active, so you can get going straight away. Next time you want to work, navigate into the *katgpucbf* directory and source the virtual environment directly:

```
source .venv/bin/activate
```

And you are ready to start developing with *katgpucbf*!

Tip: I don't recommend using the **dev-setup.sh** for anything other than initial setup. If you run it again, the requirements will be re-installed, and the module will be re-installed in editable mode. It's unlikely that any of this will be harmful in any way, but it will use up a few minutes. You probably won't want to do that every time.

12.2 Pre-commit

katgpucbf is configured with pre-commit for auto-formatting Python code. Pre-commit runs whenever anything is committed to the repository.

For more detailed information, please consult the pre-commit documentation. The installation and initialisation of the pre-commit flow is handled in **dev-setup.sh**.

12.2.1 Configuration Files

This repo contains the following configuration files for the pre-commit flow to monitor Python development.

- `.pre-commit-config.yaml` for pre-commit specifies which git hooks will be run before committing to the repo.
- `pyproject.toml` dictates the configuration of utilities such as black and isort.
- `.flake8` for flake8, a tool for enforcing **PEP 8**-based style guide for Python.
- `.pydocstyle.ini` for pydocstyle, a tool for enforcing **PEP 257**-based docstring style guides for Python.
- `mypy.ini` file for mypy, a static type checker (or lint-like tool) for type annotations in the Python code - according to **PEP 484** and **PEP 526** notation.

12.2.2 Installation Prerequisites

Although black, flake8, pydocstyle and mypy are used, the only prerequisite is the **pre-commit** Python library. That is, the YAML configuration file is set up so that when the pre-commit hooks are installed, all dependencies are automatically installed. (Note, they won't be available to you in your Python environment, they will be used only by pre-commit. If you want to use them separately, you will need to install them separately with pip.)

12.3 Light-weight installation

There are a few cases where it is unnecessary (and inconvenient) to install CUDA, such as for building the documentation or launching a correlator on a remote system. If one does not use **dev-setup.sh** but installs manually (in a virtual environment) using `pip install -e .`, then only a subset of dependencies are installed. There are also some optional extras that can be installed, such as `pip install -e ".[doc]"` to install necessary dependencies for building the documentation. Refer to `setup.cfg` to see what extras are available.

This is not recommended for day-to-day development, because it will install whatever is the latest version at the time, rather than the known-good versions pinned in requirements.txt.

12.4 Boiler-plate files

The module contains the following boiler-plate files:

- `Dockerfile` for generating repeatable container images which are capable of running this package.
- `Jenkinsfile` for a Jenkins Continuous Integration (CI) server to run unit tests automatically. Comments in the file document hardware requirements.
- `requirements.in` and `requirements-dev.in` specify the Python prerequisites for running and developing with this package respectively. They are used as inputs to pip-compile.

- `requirements.txt` and `requirements-dev.txt` list complete pinned requirements, to ensure repeatable operation. These are the output of the `pip-compile` process mentioned above. These should be passed to `pip install` with the `-r` flag to install the requirements either to run or develop. Development requires an additional set of packages which are not required for users to run the software (such as pytest). Note that developers should install both sets of requirements, not just the development ones.
- `setup.cfg` and `setup.py` allow setuptools to install this package.
- `pyproject.toml` is a standard file included with many Python projects. It is used to store some configuration for pre-commit (as described above), some configuration options for `pytest`, and other configuration as described here.

12.5 Preparing to raise a Pull Request

12.5.1 Pre-commit compliance

Contributors who prefer to develop without pre-commit enabled will be required to ensure that any submissions pass all the checks described here before they can be accepted and merged.

No judgement, we know pre-commit can be annoying if you're not used to it. This is in place in order to keep the code-base consistent so we can focus on the work at hand - rather than maintaining code readability and appearance.

12.5.2 Module documentation updates

katgpucbf holds documentation within its code-base. sphinx-apidoc provides a manner to generate module documentation as reStructuredText. If you, the developer, add or remove a module or file, execute the full `sphinx-apidoc` command below to regenerate the module documentation with your updates. The incantation below is run from the root *katgpucbf* directory.

```
sphinx-apidoc -efo doc/ src/
```

Note: The above command will likely generate a `modules.rst` file, which is not necessary to commit.

CHAPTER

THIRTEEN

UNIT TESTING

Unit testing for this module is performed using `pytest` with support from `pytest-asyncio`. Unit test files should follow `pytest` conventions. Additionally, `coverage` is used to give the developer insight into what the unit tests are actually testing, and what code remains untested. Both of these packages are installed if the `dev-setup.sh` script is used as described in *Development Environment*.

In order to run the tests, use the following command:

```
pytest
```

`pytest` reads its configuration from `pyproject.toml`. Also installed as part of this project's `requirements-dev.txt` are `coverage` and `pytest-cov`. As currently configured, running the unit tests as described above will execute a subset of the parameterised tests (see the docstring for `test/conftest.py`). While every combination of parameters won't always be tested, each individual parameter will be tested at least once.

If you'd like an HTML test-coverage report (at the expense of a slightly longer time taken to run the test), execute `pytest` with the `--cov` flag. This report can then be viewed by:

```
xdg-open htmlcov/index.html
```

Or, if you are developing on a remote server:

```
cd htmlcov && python -m http.server 8089
```

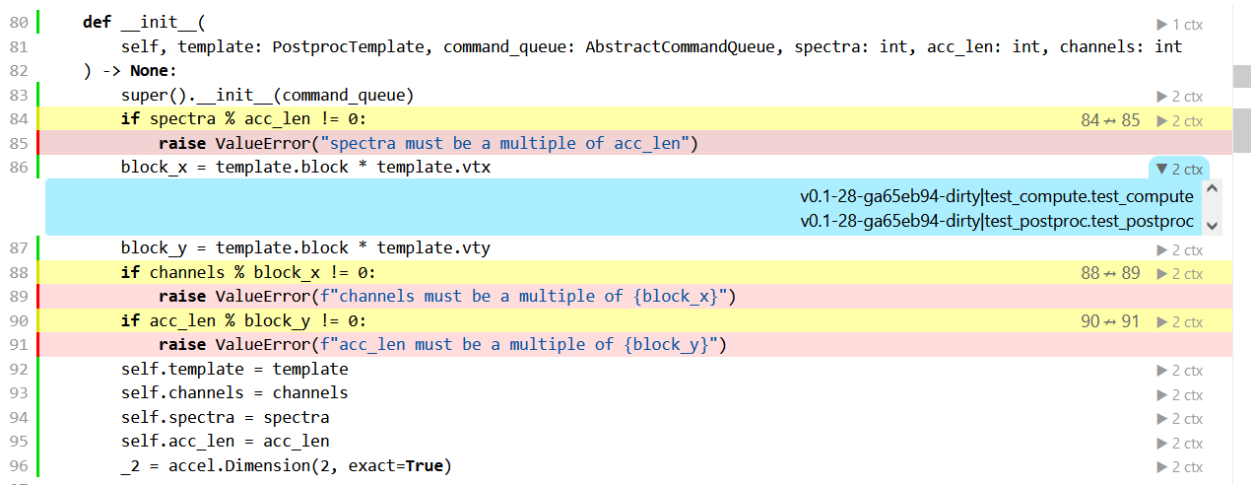
If you are using VSCode, the editor will prompt you to open the link in a browser, and automatically forward the port to your `localhost`. If not, or if you'd prefer to do it the old-fashioned way, point a browser at port `8089` on the machine that you are developing on.

The results will look something like this:

```
79
80      def __init__(                                                                                    ▶ 1 ctx
81          self, template: PostprocTemplate, command_queue: AbstractCommandQueue, spectra: int, acc_len: int, channels: int
82      ) -> None:
83          super().__init__(command_queue)                                                              ▶ 2 ctx
84          if spectra % acc_len != 0:                                                          84 ↛ 85  ▶ 2 ctx
85              raise ValueError("spectra must be a multiple of acc_len")
86          block_x = template.block * template.vtx                                                      ▶ 2 ctx
87          block_y = template.block * template.vty                                                      ▶ 2 ctx
88          if channels % block_x != 0:                                                         88 ↛ 89  ▶ 2 ctx
89              raise ValueError(f"channels must be a multiple of {block_x}")
90          if acc_len % block_y != 0:                                                          90 ↛ 91  ▶ 2 ctx
91              raise ValueError(f"acc_len must be a multiple of {block_y}")
92          self.template = template                                                                     ▶ 2 ctx
93          self.channels = channels                                                                     ▶ 2 ctx
94          self.spectra = spectra                                                                       ▶ 2 ctx
95          self.acc_len = acc_len                                                                       ▶ 2 ctx
96          _2 = accel.Dimension(2, exact=True)                                                          ▶ 2 ctx
```

The colour key is at the top of the page, but briefly, lines marked in green were executed by the tests, red were not. Yellow lines indicate branches which were only partially covered, i.e. all possible ways to branch were not tested. In the cases shown, it is because only expected values were passed to the function in question: the unit tests didn't pass invalid inputs in order to check that exceptions were raised appropriately.

On the right hand side, a context is shown for the lines that were executed, as shown in this image:



On the left side of the | is the static context - in this case showing information regarding the git commit that I ran the test on. The right side shows the dynamic context - in this case, two different tests both executed this code during the course of their run.

Note: `coverage`'s "dynamic context" output is currently specified by `pytest-cov` to describe the test function which executed the line of code in question. If desired, it can instead be specified in coverage's configuration as described in coverage's documentation. This produces a slightly different output which conveys more or less similar information.

`coverage`'s static context is more difficult to specify in a way that is useful. To generate the report above, I executed the following command:

```
coverage run --context=$(git describe --tags --dirty --always)
```

This gives more useful information about exactly what code was run, and whether it's committed or dirty. Unfortunately, doing things this way you miss out on the features of `pytest-cov`. `coverage` supports specifying a static context using either the command line (as shown) or via its configuration file, including reading of environment variables, but support doesn't extend to evaluating arbitrary shell expressions as is possible from the command line.

The package author suggests the use of a Makefile to generate an environment variable which the configuration can then use in generating a static context. This strikes me as a good solution, but I am reluctant to include yet another boiler-plate file in the repository, so I leave this to the discretion of the individual developer to make use of as desired.

Tip: Although having said that, the Makefile could also replace dev-setup.sh, allowing the developer to do something like

```
make develop  # to set up the environment
make test     # to actually run the tests
```

CHAPTER

FOURTEEN

DIGITISER PACKET SIMULATOR

The digitiser simulator (*dsim* for short) is a tool that provides the same heap format as the MeerKAT digitisers, with a configurable payload.

14.1 Usage

The dsim process generates an arbitrary number of single-pol data streams. However, it only uses a single sending thread, so in practice it does not scale well beyond two streams (a dual-pol antenna) for typical MeerKAT bandwidths. Instead, one can use multiple instances. It also relies heavily on the ibverbs support in spead2 for performance at typical MeerKAT bandwidths. It can nevertheless be used without it, but the bandwidth will most likely need to be reduced. Pass `--ibv` to use the ibverbs acceleration.

When using multiple processes, it is usually necessary to synchronise them. The `--sync-time` specifies a time (in the past) that will correspond to a zero timestamp. The synchronisation is accurate to about a millisecond, provided that all threads are pinned to specific CPU cores and real-time scheduling is used to prevent other tasks from sharing those cores. Streams sent by the same process are interleaved in a single transmit queue, so will be perfectly synchronised as they leave the NIC (but could be desynchronised by a multi-path network).

By default the content of the signal is a sine wave with a fixed frequency. However, the signal is highly configurable with the `--signals` option. A domain-specific language (DSL) allows continuous waves and Gaussian noise to be combined with basic operators (see below). The signals to send can also be changed on the fly by issuing the `?signals` command over katcp.

14.2 Signal specification

14.2.1 Basics

To specify a signal, one writes an expression followed by a semi-colon. This provides the signal for a single polarisation, so must be repeated for the number of single-pol streams. For example, the following[1] generates a continuous wave on the first polarisation and noise on the second polarisation:

```
cw(1.0, 1e9);
wgn(0.05);
```

Note that the semi-colons are required. A common mistake is to forget the final semi-colon. The following functions are available:

[1] While shown split over multiple lines, whitespace is not significant and it may be easier to place it all on one line.

cw(*amplitude*, *frequency*)
: Continuous wave with the given amplitude and frequency (in Hz). There is currently no way to directly control phase, although the `delay` function below gives limited control.

comb(*amplitude*, *frequency*)
: A comb of impulses, each one sample wide with the given amplitude. Note that if the frequency doesn't correspond to an integer number of samples, these will not be precisely periodic as each impulse time will be rounded to the nearest sample.

wgn(*std* [, *entropy*])
: White Gaussian noise with given standard deviation. Optionally, one may provide a non-negative integer seed in *entropy* to give reproducible results.

delay(*signal*, *delay*)
: Delay another signal expression by *delay* samples. For example, `delay(cw(1.0, 1e9), 10)` would shift the phase of a CW. Only integer numbers of samples are supported (including negative values).

constant
: A real number can be used as a signal, which will be used for all samples (DC).

The output magnitude is limited to the range -1 to 1, so typically the *amplitude* for `cw` and `comb` should be at most 1, and the *std* for `wgn` should be much less than 1.

14.2.2 Operators

In addition to these basic building blocks, signals can be combined with the operators +, - and *. It should be noted that these operators can only be used on signals: the scalar arguments like *amplitude* and *std* must be literal constants rather than expressions.

14.2.3 Variables

As in Python, it is also possible to assign an expression to a variable, and use the variable several times later. This has several advantages:

1. It saves typing.
2. The common part only needs to be computed once, speeding up evaluation.
3. Random choices (such as in `wgn`) are "locked in" to the variable. That simplifies creation of correlated signals without needing to explicitly choose entropy values.

As an example, the following specification defines two signals which share a sine wave and some noise, and adds further noise that is uncorrelated between the polarisations:

```
base = cw(1.0, 1e9) + wgn(0.1);
base + wgn(0.05);
base + wgn(0.05);
```

Variables can only be defined once, and must be defined before they are used. As before, statements that don't define a variable define one of the outputs, and there must be exactly one such statement per single-pol stream.

14.2.4 Dithering

By default, the signal is dithered as a final step, by adding random values uniformly selected from the interval [-0.5, 0.5) least significant bits. The dither values are chosen independently for each single-pol stream, so that they are uncorrelated.

Dithering can be disabled for an output by wrapping the expression in `nodither()`. A `nodither` signal can be assigned to a variable, but it cannot be combined with other signals using operators nor modified using `delay`.

14.3 Design

14.3.1 Signal generation

It would be extremely challenging for a CPU to simulate a signal in real-time, particularly given the need to pack the results into 10-bit samples. Instead, a window of signal is generated on startup, or on request to change the signal, and then replayed over and over. The length of this window is determined by the `--signal-heaps` command-line option. This has a few implications:

1. The frequency resolution for `cw` and `comb` is limited by the inverse of the window length. For example, a sinusoidal signal must have an integer number of cycles per window, which means that the frequency is rounded to a multiple of $\frac{\text{adc-sample-rate}}{\text{signal-heaps} \times \text{heap-samples}}$.
2. Noise is correlated in time, and when averaging over long periods of time (longer than the window), the standard deviation does not decrease with the square root of the integration time. Similarly, the sample mean converges to the mean of the generated window rather than the population mean.

To speed up the signal generation, dask is used to parallelise the process across multiple CPU cores. Dask presents a numpy-like interface, but internally splits arrays into chunks and performs computations for each chunk in parallel. The chunk size is determined by the constant *`katgpucbf.dsim.signal.CHUNK_SIZE`*.

The simulator also populates the saturation count and flag in the `digitiser_status` SPEAD item. A per-heap saturation count is computed with dask (prior to bit-packing), and then packed into the `digitiser_status` bit-field with serial code. This accumulation could be done in parallel for better efficiency, but the current approach is reasonably performant and does not require the sampling code to have any knowledge of the format of the `digitiser_status` item.

Generating reproducible random signals needs to be done carefully when parallelising. The given random seed is first used to produce a `SeedSequence` for each chunk, and each chunk then uses an independent generator seeded with its corresponding sequence. This ensures that different instances of the simulator will produce the same sequence given the same entropy (hence giving correlated noise). Note that the result is dependent on the chunk size.

14.3.2 Transmission

Most of the heavy work of transmission is handled by spead2. To minimise overheads, the heaps are pre-defined, and put into a `spead2.send.HeapReferenceList` for bulk transmission with `spead2.send.asyncio.AsyncStream.async_send_heaps()`. Additionally, spead2's rate limiting is used to control the simulated digitiser clock speed. Since spead2 sends data in small bursts (64 KiB) between sleeps, the delivery of packets will not be as smooth as from a real MeerKAT digitiser.

To avoid stalling transmission, it is important that spead2's C++ worker thread always has more data to send, as the latency of signalling end-of-transmission to Python and then waiting for Python to respond with new heaps would be significant. To accommodate this, the window is split in half, and each call to spead2 sends only half the window. As soon as one half finishes transmission, the Python code prepares it to be sent again, in parallel with spead2 starting transmission of the other half.

Although the signal is recycled, some work is still needed to prepare a half-window for retransmission, because the timestamps need to be updated. To make this as efficient as possible, all the timestamps are allocated in a single numpy array, and each heap references the appropriate entry of the array. This allows a range of timestamps to be updated with a single numpy operation, rather than a Python loop.

Allowing the signal to be changed mid-flow is done with double-buffering. The new signal is computed asynchronously into a spare second window. Once that's completed, the spare and active windows are swapped. The new spare window may still be referenced by in-flight heaps, so it is necessary to await transmission of those heaps before allowing the signal to be changed again.

CHAPTER

FIFTEEN

F-ENGINE PACKET SIMULATOR

In general an XB-Engine needs to receive data for a subset of channels from N F-Engines where N is the telescope array size. This is complicated to configure and requires many F-Engines. In order to bypass this, an F-Engine simulator has been created that simulates packets received at the XB-Engine (i.e., packets from multiple F-Engines destined for the same XB-Engine). This simulator benefits from a server with a Mellanox NIC and ibverbs to run. This fsim simulates the packet format used by katgpucbf.

The minimum command to run fsim is:

```
fsim --interface <interface_name> <multicast_address>[+y]:<port>
```

where

- *<interface_name>* is the name of the network interface on which to transmit the data;
- *<multicast_address>* is the multicast address to which all packets are sent. The optional *[+y]* argument will create additional multicast streams with the same parameters each on a different multicast addresses consecutivly after the base address. *<port>* is the UDP port to transmit data to.

The data rate per multicast address is adc_rate * N_POLS * SAMPLE_BITS * antennas * (channels_per_substream / channels). With the default arguments, this is 1712000000 * 2 * 8 * 80 * (512/32768) = 34.24 Gbps.

For improved performance, use `--ibv` to enable ibverbs acceleration[1]. This requires the `CAP_NET_RAW` capability to run. The easiest way to do it is with `spead2_net_raw`.

See the fsim source code (in `src/katgpucbf/fsim/`) for a detailed description of how the F-Engine simulator works and the useful configuration arguments.

[1] See the spead2 documentation for information on the requirements (particularly hardware requirements) for ibverbs.

CHAPTER

SIXTEEN

QUALIFICATION FRAMEWORK

While the unit tests ensure that individual pieces of the CBF work correctly, the qualification tests ensure that the system as a whole functions correctly and meets requirements. In software engineering terms these are integration tests.

The qualification tests are stored in the `qualification` directory and are run with pytest. In addition to the usual pass or fail indication, the tests produce a report (in PDF format), which describes which tests were run, the steps involved in the tests, the machines used and so on. It also includes assorted plots showing results.

The tests do not run the katgpucbf code from the local machine. Instead, they connect to an SDP Master Controller and use it to start appropriate CBFs which they interact with. Facilities are provided for the test to interact with the CBF, both by sending it KATCP requests and by ingesting the output data. It's thus necessary to have a master controller set up (which is beyond the scope of this document) and to have a Docker image of katgpucbf stored in a Docker registry.

Additionally, the hosts in the cluster must be monitored by Prometheus, so that the qualification report can include information on the hardware and software configuration. They must run node-exporter with the arguments `--collector.cpu.info` and `--collector.ethtool`.

16.1 Requirements

A `requirements.in` and `requirements.txt` are provided in this directory, based on katgpucbf's `requirements-dev.txt`. A pared-down version of this may become available in future.

It's necessary to have `katgpucbf` installed for the qualification tests to run, but it is not necessary to have a GPU or CUDA installed. The necessary parts can be installed with

```
pip install ".[qualification]"
```

The machine running the tests needs to be able to receive data from the CBF network. The data rate can become quite high for larger array sizes.

16.2 Configuration

You will need to create a `qualification/pytest.ini` file. It is specific to your test environment, so do not commit it to git. You'll need to set it up only once per machine that you're deploying on, and it'll look something like this:

```
[pytest]
tester = Your Name
asyncio_mode = auto
master_controller_host = lab5.sdp.kat.ac.za
master_controller_port = 5001
```

(continues on next page)

(continued from previous page)

```
prometheus_url = http://lab5.sdp.kat.ac.za:9090
product_name = bobs_qualification_cbf  # Use your own name
interface = enp193s0f0
interface_gbps = 90  # Maximum bandwidth to expect from the NIC
use_ibv = true
log_cli = true
log_cli_level = info
addopts = --report-log=report.json
```

Only set `use_ibv` if the NIC and the system support ibverbs. See the spead2 documentation for advice on setting that up. This will probably be needed to successfully test large numbers of channels or antennas.

16.3 Running

Use the following command to run the tests contained in this directory:

```
spead2_net_raw pytest -v qualification --image-override katgpucbf:harbor.sdp.kat.ac.za/
↪dpp/katgpucbf:latest
```

Explanation:

- `spead2_net_raw` enables ibverbs usage (see `use_ibv` above)
- `--image-override` is designed to work in exactly the same way as that in `sim_correlator.py`, specifying exactly which Docker image to use for the tests.

The general pytest options apply, so for instance with `-x` you can stop after the first failed test instead of continuing, etc.

16.4 Post-processing

The steps above produce a `report.json` file. To turn that into a usable PDF, run

```
qualification/report/generate_pdf.py report.json report.pdf
```

This requires at least `texlive-base`, `texlive-latex-extra`, `texlive-science` and `latexmk`. This step doesn't interact with the live system at all, so it is possible to copy/mount the JSON file to another machine to run this step.

CHAPTER

SEVENTEEN

UPDATING AUTOTUNING DATABASE

Some kernels having tuning parameters which can be optimised automatically. Support for this is provided by the katsdpsigproc library.

To avoid autotuning occurring at startup, a database of pre-tuned parameters is stored in the repository and inserted into the Docker image. If necessary, it can be updated using the script `docker/autotune.py`, run on a machine with a suitable GPU. It does not need to exactly match the GPU used for deployment, but the more similar it is, the better the tuning will be.

To ensure that the tuning is done with the same environment (particularly CUDA compiler version) that will be deployed, the autotuning should be run inside the Docker image. This can be done something like this (the Docker image path is just an example; adjust as necessary), and takes roughly 30 minutes:

```
docker pull harbor.sdp.kat.ac.za/dpp/katgpucbf
docker run -it --rm --gpus=all -v $PWD/docker:/output harbor.sdp.kat.ac.za/dpp/katgpucbf␣
↪/output/autotune.py /output/tuning.db
```

Note that this may cause the database (`docker/tuning.db`) to be owned by root and require fixing. Check that the database is non-empty, then commit it.

The tuning script covers a reasonably large range of parameters, but cannot cover everything. If you see run-time log messages indicating that autotuning is occurring, you may need to revise the script.

CHAPTER

EIGHTEEN

BENCHMARKING

Performing a benchmark of fgpu or xbgpu under realistic conditions (in particular, with data transmitted over UDP) is difficult because there is no flow control, and the achievable rate can only be observed indirectly by running the system at a particular rate and checking whether it keeps up. We wish to know the largest rate at which we keep up most of the time; we'll call this the "critical" rate. The `scratch/benchmarks` directory contains a script to help estimate the critical rate for fgpu (xbgpu support may be added later).

To run it, you'll need

- two servers (one to generate data and one to receive it)
  - configured for use with spead2's ibverbs support on a high-speed network between them;
  - with a Docker daemon;
- somewhere to run the script (referred to below as the "client"), set up for key-based SSH access to the servers in accounts that can invoke the Docker client.
- a Docker registry containing the katgpucbf Docker image.

The assignment of threads to processor cores on the servers has been optimised for AMD Epyc servers with at least 16 cores; for other servers or for lower core counts, the benchmark script may need to be tuned to provide sensible placement.

The client machine will also need a few Python packages installed; you can use **`pip install -r scratch/benchmarks/requirements.txt`** to install them. You do not need to have katgpucbf installed.

On the client machine, you will need to create a TOML file describing the servers you want to use. Each table in the TOML file describes one server. You can give them any names you like, but by default the script will use the servers called `dsim` and `fgpu`. A typical file looks like this:

```
[dsim]
hostname = 'server01.domain'
username = 'myusername'
interfaces = ['enp193s0f0np0', 'enp193s0f1np1']

[fgpu]
hostname = 'server02.domain'
username = 'myusername'
interfaces = ['enp193s0f0np0', 'enp193s0f1np1']
```

The *interfaces* arrays list the names of the ibverbs-capable network interfaces that can be used for sending or receiving the data.

By default the servers are loaded from `servers.toml` in the current directory, although the *`--servers`* command-line option can override this.

With all the requirements in place, change to the `scratch/benchmarks` directory and run **`./benchmark_fgpu.py`**. There are some options you may need:

`-n` <N>

Set the number of engines to run simultaneously. The benchmark has been developed for values 1 and 4, and may need further tuning to effectively test other values.

`--image` <image>

Override the Docker image to run. The default is set up for the lab environment at SARAO.

`--low` <rate>, **`--high`** <rate>

Lower and upper bounds on the ADC sample rate. The critical rate will be searched between these two bounds. The benchmark will error out if the lower bound fails or the upper bound passes.

`--interval` <rate>

Target confidence interval size. The final result will not be an exact rate (because the process is probabilistic) but rather a range. The algorithm will keep running until it is almost certain that the critical rate is inside an interval of this size or less. Setting this smaller than roughly 1% of the critical rate can cause the algorithm to fail to converge (because there is too much noise to determine the rate more accurately). See also *--max-comparisons*.

`--max-comparisons` <N>

Bound the number of comparisons to perform in the search (excluding the sanity checks that the low and high rates are reasonable bounds). If *--interval* is too small, the algorithm might not otherwise converge. If this number of comparisons is exceeded, a larger interval will be output.

`--step` <rate>

Only multiples of this value will be tested. Some parts of the algorithm require time proportional to the square of the number of steps between the low and high rate, so this should not be too small. However, it must not be larger than *--interval*.

`--servers` <filename>

Server description TOML file.

`--dsim-server` <name>, **`--fgpu-server`** <name>

Override the server names to find in the servers file.

This is not a complete list of options; run the command with `--help` to see others.

18.1 Multicast groups

The benchmark code currently hard-codes a number of multicast groups. Thus, **two instances cannot be run on the same network at the same time**. The groups are all in the 239.102.0.0/16 subnet.

18.2 Algorithm

The algorithm can be seen as a noisy binary search using Bayesian inference. Even when running at a slow enough rate, packets may be randomly lost, and this could send a classical binary search down the wrong side of the search tree and yield a very incorrect answer.

Instead, the range from *--low* to *--high* is divided into bins of size *--step*, and for each bin, we keep a probability that the true rate falls into that bin. We also have a model of how likely a trial is to succeed at a given rate, if we know the critical rate: very likely/unlikely if the given rate is significantly lower/higher than the critical rate, and more uncertain when the given rate is close to critical. After running a trial, we can use Bayes' Theorem to update the probability

distribution. To choose the binary boundary to test, we consider every option and pick the one that gives the largest expected decrease in the entropy of the distribution.

Determining the success model is non-trivial and an incorrect model could lead to inaccurate answers (as a simple example, a model that considers trials to be perfect would reduce to classical binary search, which as already discussed is problematic). The benchmark script also supports a "calibration" mode, in which every candidate rate is tested a large number of times and the fraction of successes is printed. This does not automatically feed this information back into the (hard-coded) model.

--calibrate

Run the calibration mode instead of the usual search

--calibrate-repeat <N>

Set the number of repetitions for each rate.

It is highly recommended that *--low*, *--high* are used to specify a much smaller range around the critical rate, as this process is extremely slow.

The output of this calibration process is a text file of space-separated values. Previously-collected results are in the `fgpu_benchmarks` subdirectory, and new additions should go here too. After adding or updating one of these files, run **./fit.py** and pass it the filename. It will print out the coefficients for a fitted logistic regression model. The key information is the `np.log(rate)` term, which can then be stored in the `slope` variable in `benchmark.py`. You can also pass `--plot` to **./fit.py** to get a plot of the calibration results versus the fitted model (requires matplotlib).

CHAPTER

NINETEEN

TODOS

This list is assembled from throughout the documentation. If you're looking for something to keep yourself busy, this is a good place to start.

Note: This list only includes TODOs formatted in a way that Sphinx understands. There are likely others formatted as comments throughout the code which don't appear listed here. `grep` can help you find them!

The `test` and `qualification` folders are not pulled in by Sphinx, and so any TODOs there will also not be included in this list.

Todo: `NGC-730` Update scratch directory to have a single config sub-directory. Also add comments on the scripts themselves to make it easier to follow.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/katgpucbf/checkouts/latest/doc/control.rst, line 125.)

Todo: `NGC-730` Update `run-{dsim, fpgu, xbgpu}.sh` scripts to standardise over usage of either `numactl` or `taskset`.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/katgpucbf/checkouts/latest/doc/control.rst, line 179.)

Todo: If this section gets to be too large, it can probably also make its way into its own file.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/katgpucbf/checkouts/latest/doc/data_interfaces.rst, line 4.)

Todo: `NGC-680` - Relationship with katsdpcontroller - reference to a later section which will describe it more thoroughly.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/katgpucbf/checkouts/latest/doc/introduction.rst, line 89.)

Todo: Eventually modify the classes to support 4 and 16 bit input samples. The kernel supports this, but it is not exposed to the reader. There is no use case for this at the moment, so this is a low priority.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/katgpucbf/envs/latest/lib/python3.10/site-packages/katgpucbf/xbgpu/correlation.py:docstring of katgpucbf.xbgpu.correlation, line 3.)

Todo: Document the down-conversion filter

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/katgpucbf/checkouts/latest/doc/maths.rst, line 66.)

CHAPTER

TWENTY

KATGPUCBF PACKAGE

20.1 Subpackages

20.1.1 katgpucbf.dsim package

Submodules

katgpucbf.dsim.descriptors module

Digitiser simulator descriptor sender.

katgpucbf.dsim.descriptors.**create_config**() → StreamConfig

Create simple configuration for descriptor stream.

katgpucbf.dsim.descriptors.**create_descriptors_heap**() → Heap

Create a descriptor heap for output dsim data.

katgpucbf.dsim.main module

Digitiser simulator.

Simulates the packet structure of MeerKAT digitisers.

async katgpucbf.dsim.main.**async_main**() → None

Asynchronous main entry point.

katgpucbf.dsim.main.**first_timestamp**(*time_converter:* TimeConverter, *now: float*, *align: int*) → int

Determine ADC timestamp for first sample and the time at which to start sending.

The resulting value will be a multiple of *align*.

Parameters

- **time_converter** – Time converter between UNIX timestamps and ADC samples
- **now** – Lower bound on first timestamp, expressed as UNIX timestamp
- **align** – Alignment requirement on the returned ADC sample count

katgpucbf.dsim.main.**main**() → None

Run main program.

katgpucbf.dsim.main.**parse_args**(*arglist: Sequence[str] | None = None*) → Namespace

Parse the command-line arguments.

katgpucbf.dsim.send module

Transmission of SPEAD data.

class katgpucbf.dsim.send.**HeapSet**(*data: Dataset*)

Bases: object

Collection of heaps making up a signal.

The heaps are split into two parts, each of which is preprocessed to allow efficient transmission.

This class should normally be constructed with `factory()`.

Parameters

data – An xarray data set with the following variables:

timestamps
1D array of timestamps, big-endian 64-bit

digitiser_status
2D array of digitiser status values, big-endian 64-bit (indexed by polarisation and time)

payload
2D array of raw sample data (indexed by polarisation and time)

heaps
Heaps referencing the timestamps and payload

The dimensions must be `time`, `pol` and `data`.

classmethod create(*timestamps: ndarray*, *n_substreams: Sequence[int]*, *heap_size: int*, *digitiser_id: Sequence[int]*) → *HeapSet*

Create from shape parameters.

Parameters

- **timestamps** – The timestamp array to associate with the *HeapSet* (must be big-endian 64-bit).
- **n_substreams** – Number of substreams to distribute the heaps across, per polarisation
- **heap_size** – Number of bytes of payload per heap
- **digitiser_id** – Digitiser ID to insert into the packets, per polarisation (LSB should indicate polarisation)

class katgpucbf.dsim.send.**Sender**(*stream: spead2.send.asyncio.AsyncStream*, *heap_set:* HeapSet, *heap_samples: int*)

Bases: object

Manage sending packets.

halt() → None

Request *run()* to stop, but do not wait for it.

async join() → None

Wait for *run()* to finish.

This does not cause it to stop: use *halt()* for that.

async run(*first_timestamp: int*, *time_converter:* TimeConverter) → None

Send heaps continuously.

async set_heaps(*heap_set:* HeapSet) → int

Switch out the heap set for a different one.

This does not return until the payload of the previous *HeapSet* is no longer in use (the timestamps may still be in use).

The new heap_set must share timestamps with the old one.

Returns
First timestamp which will use the new heap set

Return type
timestamp

async stop() → None

Stop *run()* and wait for it to finish.

katgpucbf.dsim.send.**make_stream**(*, *endpoints: Iterable[tuple[str, int]]*, *heap_sets: Iterable[*HeapSet*]*, *n_pols: int*, *adc_sample_rate: float*, *heap_samples: int*, *sample_bits: int*, *max_heaps: int*, *ttl: int*, *interface_address: str*, *ibv: bool*, *affinity: int*) → spead2.send.asyncio.AsyncStream

Create a spead2 stream for sending.

Parameters

- **endpoints** – Destinations (host and port) for all substreams
- **n_pols** – Number of single-pol streams to send
- **adc_sample_rate** – Sample rate for each single-pol stream, in Hz
- **heap_samples** – Number of samples to send in each heap (each heap will be sent as a single packet)
- **sample_bits** – Number of bits per sample
- **max_heaps** – Maximum number of heaps that may be in flight at once
- **ttl** – IP TTL field
- **interface_address** – IP address of the interface from which to send the data
- **ibv** – If true, use ibverbs for acceleration
- **affinity** – If non-negative, bind the sending thread to this CPU core

katgpucbf.dsim.send.**make_stream_base**(*, *config: StreamConfig*, *endpoints: Iterable[tuple[str, int]]*, *ttl: int*, *interface_address: str*, *ibv: bool = False*, *affinity: int = -1*, *memory_regions: list | None = None*) → spead2.send.asyncio.AsyncStream

Create a spead2 stream for sending.

This is the low-level support for making either a data or a descriptor stream. Refer to *make_stream()* for explanations of the arguments.

katgpucbf.dsim.server module

katcp server.

class katgpucbf.dsim.server.**DeviceServer**(*sender: Sender*, *descriptor_sender: DescriptorSender*, *spare: HeapSet*, *adc_sample_rate: float*, *sample_bits: int*, *dither_seed: int*, **args*, ***kwargs*)

Bases: DeviceServer

katcp server.

Parameters

- **sender** – Sender which is streaming data out. It is halted when the server is stopped.
- **spare** – Heap set which is not currently being used, but is available to swap in
- **adc_sample_rate** – Sampling rate in Hz
- **sample_bits** – Number of bits per output sample
- **dither_seed** – Dither seed (used only to populate a sensor).
- ***args** – Passed to base class
- ****kwargs** – Passed to base class

BUILD_STATE: str = '0.1.dev290+gf385556'

VERSION: str = 'katgpucbf-dsim-0.1'

async on_stop() → None

Extension point for subclasses to run shutdown code.

This is called after the TCP server has been shut down and all in-flight requests have been completed or cancelled, but before service tasks are cancelled. Subclasses should override this function rather than `stop()` to run late shutdown code because this is called *before* the flag is set to wake up `join()`.

It is only called if the server was running when `stop()` was called.

async request_signals(*ctx*, *signals_str: str*, *period: int | None = None*) → int

Update the signals that are generated.

Parameters

- **signals_str** – Textural description of the signals. See the docstring for parse_signals for the language description. The description must produce one signal per polarisation.
- **period** – Period for the generated signal. It must divide into the value indicated by the `max-period` sensor. If not specified, the value of `max-period` is used.

Returns
First timestamp which will use the new signals

Return type
timestamp

async request_time(*ctx*) → float

Return the current UNIX timestamp.

async set_signals(*signals: Sequence[Signal]*, *signals_str: str*, *period: int | None = None*) → int

Change the signals *request_signals()*.

This is the implementation of *request_signals()*. See that method for description of the parameters and return value (*signals* is the parsed version of *signals_str*).

katgpucbf.dsim.shared_array module

Shared memory arrays.

class katgpucbf.dsim.shared_array.**SharedArray**(*fd: int*, *shape: tuple[int, ...]*, *dtype: dtype[Any] | None | type[Any] | _SupportsDType[dtype[Any]] | str | tuple[Any, int] | tuple[Any, SupportsIndex | Sequence[SupportsIndex]] | list[Any] | _DTypeDict | tuple[Any, Any]*)

Bases: object

An array that can be passed to another process.

Unlike multiprocessing.shared_memory, the shared memory used for this is not backed by a file, and so is guaranteed to be cleaned up when the processes involved die off, without the need for a manager process.

This is UNIX (probably Linux) specific.

Do not construct directly. Instead, either use *create()* to allocate a new array, or multiprocessing.connection.Connection.recv() to construct a new reference to an existing array in another process.

close() → None

Close the reference shared array and release the mapping.

Accessing the array after this will most likely crash. It is safe to call twice.

classmethod create(*name: str*, *shape: tuple[int, ...]*, *dtype: dtype[Any] | None | type[Any] | _SupportsDType[dtype[Any]] | str | tuple[Any, int] | tuple[Any, SupportsIndex | Sequence[SupportsIndex]] | list[Any] | _DTypeDict | tuple[Any, Any]*) → *SharedArray*

Create a new array from scratch.

Parameters

- **name** – An arbitrary name to associate with the array. See os.memfd_create().
- **shape** – Shape of the array. To simplify this function, it requires a tuple (a scalar cannot be used).
- **dtype** – The type of the array.

katgpucbf.dsim.signal module

Synthesis of simulated signals.

katgpucbf.dsim.signal.**CHUNK_SIZE = 1048576**

Dask chunk size for sampling signals (must be a multiple of 8)

class katgpucbf.dsim.signal.**CW**(*amplitude: float*, *frequency: float*)

Bases: *Periodic*

Continuous wave.

To make the resulting signal periodic, the frequency is adjusted during sampling so that the sampled result can be looped.

class katgpucbf.dsim.signal.**Comb**(*amplitude: float*, *frequency: float*)

Bases: *Periodic*

Signal with periodic impulses.

To make the resulting signal periodic, the frequency is adjusted during sampling so that the sampled result can be looped.

class katgpucbf.dsim.signal.**CombinedSignal**(*a: Signal*, *b: Signal*, *combine: Callable[[Array, Array], Array]*, *op_name: str*)

Bases: *Signal*

Signal built by combining two other signals.

Parameters

- **a** – Input signals
- **b** – Input signals
- **combine** (*collections.abc.Callable[[dask.array.core.Array, dask.array.core.Array], dask.array.core.Array]*) – Operator to combine two arrays
- **op_name** (*str*) – Symbol for the operator

a: *Signal*

b: *Signal*

combine: Callable[[Array, Array], Array]

op_name: str

sample(*n: int*, *sample_rate: float*) → Array

Sample the signal at regular intervals.

The returned values should be scaled to the range (-1, 1).

Note: Calling this method with two different values of *n* may yield results that are not consistent with each other.

Parameters

- **n** – Number of samples to generate
- **sample_rate** – Frequency of samples (Hz)

Returns
Dask array of samples, float32. The chunk size must be CHUNK_SIZE.

Return type
samples

class katgpucbf.dsim.signal.**Constant**(*value: float*)

Bases: *Signal*

Fixed value.

sample(*n: int*, *sample_rate: float*) → Array

Sample the signal at regular intervals.

The returned values should be scaled to the range (-1, 1).

Note: Calling this method with two different values of *n* may yield results that are not consistent with each other.

Parameters

- **n** – Number of samples to generate
- **sample_rate** – Frequency of samples (Hz)

Returns
Dask array of samples, float32. The chunk size must be CHUNK_SIZE.

Return type
samples

value: float

class katgpucbf.dsim.signal.**Delay**(*signal:* Signal, *delay: int*)

Bases: *Signal*

Delay another signal by an integer number of samples.

Parameters

- **signal** (katgpucbf.dsim.signal.Signal) – Underlying signal to delay
- **delay** (*int*) – Number of samples to delay the signal (may be negative)

delay: int

sample(*n: int*, *sample_rate: float*) → Array

Sample the signal at regular intervals.

The returned values should be scaled to the range (-1, 1).

Note: Calling this method with two different values of *n* may yield results that are not consistent with each other.

Parameters

- **n** – Number of samples to generate
- **sample_rate** – Frequency of samples (Hz)

Returns
Dask array of samples, float32. The chunk size must be CHUNK_SIZE.

Return type
samples

signal: *Signal*

class katgpucbf.dsim.signal.**Nodither**(*signal:* Signal)

Bases: *Signal*

Mark a signal expression as not needing dither.

Parameters
signal (katgpucbf.dsim.signal.Signal) – Underlying signal

sample(*n: int*, *sample_rate: float*) → Array

Sample the signal at regular intervals.

The returned values should be scaled to the range (-1, 1).

Note: Calling this method with two different values of *n* may yield results that are not consistent with each other.

Parameters

- **n** – Number of samples to generate
- **sample_rate** – Frequency of samples (Hz)

Returns
Dask array of samples, float32. The chunk size must be CHUNK_SIZE.

Return type
samples

signal: *Signal*

property terminal: bool

Prevent this signal from being used in expressions.

class katgpucbf.dsim.signal.**Periodic**(*amplitude: float*, *frequency: float*)

Bases: *Signal*

Base class for period signals.

The frequency is adjusted during sampling so that the sampled result can be looped.

amplitude: float

frequency: float

sample(*n: int*, *sample_rate: float*) → Array

Sample the signal at regular intervals.

The returned values should be scaled to the range (-1, 1).

Note: Calling this method with two different values of *n* may yield results that are not consistent with each other.

Parameters

- **n** – Number of samples to generate

- **sample_rate** – Frequency of samples (Hz)

Returns
Dask array of samples, float32. The chunk size must be CHUNK_SIZE.

Return type
samples

class katgpucbf.dsim.signal.**Random**(*entropy: int | None = None*)

Bases: *Signal*

Base class for randomly-generated signals.

This base class is only suitable when the samples at different times are independent. The derived class must implement `_sample_chunk()`.

entropy: int

entropy used to populate a `np.random.SeedSequence`

sample(*n: int*, *sample_rate: float*) → Array

Sample the signal at regular intervals.

The returned values should be scaled to the range (-1, 1).

Note: Calling this method with two different values of *n* may yield results that are not consistent with each other.

Parameters

- **n** – Number of samples to generate
- **sample_rate** – Frequency of samples (Hz)

Returns
Dask array of samples, float32. The chunk size must be CHUNK_SIZE.

Return type
samples

class katgpucbf.dsim.signal.**Signal**

Bases: ABC

Abstract base class for signals.

An instance is simply a real-valued function of time, for a single polarisation.

abstract sample(*n: int*, *sample_rate: float*) → Array

Sample the signal at regular intervals.

The returned values should be scaled to the range (-1, 1).

Note: Calling this method with two different values of *n* may yield results that are not consistent with each other.

Parameters

- **n** – Number of samples to generate

- **sample_rate** – Frequency of samples (Hz)

Returns
Dask array of samples, float32. The chunk size must be CHUNK_SIZE.

Return type
samples

property terminal: bool

Indicate whether the signal is terminal.

Terminal signals cannot be combined into larger expressions, because they contain information about how to handle their postprocessing.

class katgpucbf.dsim.signal.**SignalService**(*arrays: Sequence[DataArray]*, *sample_bits: int*, *dither_seed: int | None = None*)

Bases: object

Compute signals in a separate process.

The provided arrays must be backed by *SharedArray*, and each must have an xarray attribute called "shared_array" which holds the backing *SharedArray*.

Parameters

- **arrays** – All the arrays that might be passed to *sample()*.
- **sample_bits** – Number of bits per sample for all queries.
- **dither_seed** – Seed used to generate a fixed dither.

async sample(*signals: Sequence[Signal]*, *timestamp: int*, *period: int | None*, *sample_rate: float*, *out: DataArray*, *out_saturated: DataArray | None = None*, *saturation_group: int = 1*) → None

Perform signal sampling in the remote process.

out and *out_saturated* must each be one of the arrays passed to the constructor. Only the first *n* samples will be populated (and this will be taken as the period).

async stop() → None

Shut down the process.

exception katgpucbf.dsim.signal.**TerminalError**(*signal: Signal*)

Bases: TypeError

Indicate that a terminal signal has been used in an expression.

class katgpucbf.dsim.signal.**WGN**(*std: float*, *entropy: int | None = None*)

Bases: *Random*

White Gaussian Noise signal.

Each sample in time is an independent Gaussian random variable with zero mean and a given standard deviation.

In practice, the signal has a period equal to the value of *n* given to *sample()*, which could lead to undesirable correlations.

Parameters

- **std** (*float*) – Standard deviation of the samples
- **entropy** (*int*) – If provided, used to seed the random number generator

std: float = 1.0

standard deviation

katgpucbf.dsim.signal.**format_signals**(*signals: Sequence[*Signal*]*) → str

Inverse of *parse_signals()*.

Currently object identity is not preserved, so if a simple signal is re-used multiple times (e.g., shared across output signals), it will be repeated in the output. This is subject to change.

katgpucbf.dsim.signal.**make_dither**(*n_pols: int*, *n: int*, *entropy: int | None = None*) → DataArray

Create a set of dither signals to use with *quantise()*.

The returned array has `pol` and `data` axes, and is backed by a Dask array.

The implementation currently uses a uniform distribution, but that is subject to change.

katgpucbf.dsim.signal.**packbits**(*data: Array*, *bits: int*) → Array

Pack integers into bytes.

The least-significant *bits* bits of each integer in *data* is collected together in big-endian order, and returned as a sequence of bytes. The total number of bits must form a whole number of bytes.

If the chunks in *data* are not be aligned on byte boundaries then a slower path is used.

katgpucbf.dsim.signal.**parse_signals**(*prog: str*) → list[*Signal*]

Generate a set of signals from a domain-specific language.

See *Signal specification* for a description of the language.

katgpucbf.dsim.signal.**quantise**(*data: Array*, *bits: int*, *dither: Array*) → Array

Convert floating-point data to fixed-point.

Parameters

- **data** – Array of values, nominally in the range -1 to 1 (values outside the range are clamped).
- **bits** – Total number of bits per output sample (including the sign bit). The input values are scaled by $2^{bits-1} - 1$.
- **dither** – Values to add to the data after scaling.

katgpucbf.dsim.signal.**sample**(*signals: Sequence[*Signal*]*, *timestamp: int*, *period: int | None*, *sample_rate: float*, *sample_bits: int*, *out: DataArray*, *out_saturated: DataArray | None = None*, *saturation_group: int = 1*, ***, *dither: bool | DataArray = True*, *dither_seed: int | None = None*) → None

Sample, quantise and pack a set of signals.

The number of samples to generate is determined from the output array.

Parameters

- **signals** – Signals to sample, one per polarisation
- **timestamp** – Timestamp for the first element to return. The signal is rotated by this amount.
- **period** – Number of samples after which to repeat. This must divide into the total number of samples to generate. If not specified, uses the total number of samples.
- **sample_rate** – Passed to *Signal.sample()*
- **sample_bits** – Passed to *quantise()* and *packbits()*
- **out** – Output array, with a dimension called `pol` (which must match the number of signals). The other dimensions are flattened.

- **out_saturated** – Output array, with the same shape as `out`, into which saturation counts are written.
- **saturation_group** – Samples are taken in contiguous groups of this size and each element of out_saturated is a saturation count for one group. This must divide into the total number of samples.
- **dither** – If true (default), add uniform random values in the range [-0.5, 0.5) after scaling to reduce artefacts. It may also be a `xr.DataArray` with axes called `pol` (which must match the number of signals) and `data` (which must have length at least equal to `period`).

katgpucbf.dsim.signal.**saturation_counts**(*data: Array*, *saturation_value*) → Array

Return an array indicating counts of saturated elements of `data`.

The count is taken along each row of `data`.

Elements are considered saturated if they exceed *saturation_value* in absolute value.

Module contents

20.1.2 katgpucbf.fgpu package

Submodules

katgpucbf.fgpu.compute module

The *Compute* class specifies the sequence of operations on the GPU.

Allocations of memory for input, intermediate and output are also handled here.

class katgpucbf.fgpu.compute.**Compute**(*template: ComputeTemplate*, *command_queue: AbstractCommandQueue*, *samples: int*, *spectra: int*, *spectra_per_heap: int*)

Bases: `OperationSequence`

The DSP processing pipeline handling F-engine operation.

The actual running of this operation isn't done through the `_run()` method or by calling it directly, if you're familiar with the usual method of composing operations. Fgpu's compute is streaming rather than batched, i.e. we have to coordinate the receiving of new data and the transmission of processed data along with the actual processing operation.

Currently, no internal checks for consistency of the parameters are performed. The following constraints are assumed, Bad Things(TM) may happen if they aren't followed:

- spectra_per_heap <= spectra - i.e. a chunk of data must be enough to send out at least one heap.
- spectra % spectra_per_heap == 0
- samples >= output.window (see *fgpu.output.Output*). An input chunk requires at least enough samples to output a single spectrum.
- samples % 8 == 0

Parameters

- **template** – Template for the channelisation operation sequence.
- **command_queue** – The GPU command queue (typically this will be a CUDA Stream) on which actual processing operations are to be scheduled.

- **samples** – Number of samples in each input chunk (per polarisation), including padding samples.
- **spectra** – Number of spectra in each output chunk.
- **spectra_per_heap** – Number of spectra to send in each output heap.

bind(***kwargs*) → None

Bind buffers to slots by keyword.

Each keyword argument name specifies a slot name.

Raises

- **KeyError** – if a named slots does not exist
- **TypeError** – if a named slot if an alias slot

buffer(*name: str*) → DeviceArray

Retrieve the buffer bound to a slot.

It will consult both `slots` and `hidden_slots`.

Parameters
name (*str*) – Name of the slot to access

Returns
Buffer bound to slot *name*.

Return type
`DeviceArray`

Raises

- **KeyError** – If no slot with this name exists
- **TypeError** – If the slot exists but it is an alias slot
- **ValueError** – If the slot exists but does not yet have a buffer bound

ensure_all_bound() → None

Make sure that all slots have a buffer bound, allocating if necessary.

ensure_bound(*name: str*) → None

Make sure that a specific slot has a buffer bound, allocating if necessary.

parameters() → Mapping[str, Any]

Return dictionary of configuration options for this operation.

required_bytes() → int

Return number of bytes of device storage required.

run_backend(*out: DeviceArray*, *saturated: DeviceArray*) → None

Run the FFT and postproc on the data which has been PFB-FIRed.

Postproc incorporates fine-delay, requantisation and corner-turning.

Parameters
out – Destination for the processed data.

run_ddc(*samples: DeviceArray*, *first_sample: int*) → None

Run the narrowband DDC kernel on the received samples.

Parameters

- **samples** – A device array containing the samples.
- **first_sample** – Timestamp (in samples) of the initial sample. This is used to correctly phase the mixer.

run_narrowband_frontend(*in_offsets: Sequence[int]*, *out_offset: int*, *spectra: int*) → None

Run the PFB-FIR on the received samples, for a narrowband pipeline.

Parameters

- **in_offsets** – Index of first sample in input array to process (one for each pol).
- **out_offset** – Index of first sample in output array to write.
- **spectra** – How many spectra worth of samples to push through the PFB-FIR.

run_wideband_frontend(*samples: DeviceArray*, *dig_total_power: DeviceArray*, *in_offsets: Sequence[int]*, *out_offset: int*, *spectra: int*) → None

Run the PFB-FIR on the received samples, for a wideband pipeline.

Parameters

- **samples** – A device arrays containing the samples
- **dig_total_power** – A device array holding digitiser total power for each pol. This is not zeroed.
- **in_offsets** – Index of first sample in input array to process (one for each pol).
- **out_offset** – Index of first sample in output array to write.
- **spectra** – How many spectra worth of samples to push through the PFB-FIR.

class katgpucbf.fgpu.compute.**ComputeTemplate**(*context: AbstractContext*, *taps: int*, *channels: int*, *dig_sample_bits: int*, *out_bits: int*, *narrowband: NarrowbandConfig | None*)

Bases: object

Template for the channelisation operation sequence.

The reason for doing things this way can be read in the relevant katsdpsigproc docs.

Parameters

- **context** – The GPU context that we'll operate in.
- **taps** – The number of taps that you want the resulting PFB-FIRs to have.
- **channels** – Number of output channels into which the input data will be decomposed.
- **dig_sample_bits** – Number of bits per digitiser sample.
- **out_bits** – Number of bits per output real component.
- **narrowband** – Configuration for narrowband operation. If None, wideband is assumed.

instantiate(*command_queue: AbstractCommandQueue*, *samples: int*, *spectra: int*, *spectra_per_heap: int*) → *Compute*

Generate a *Compute* object based on the template.

class katgpucbf.fgpu.compute.**NarrowbandConfig**(*decimation: int*, *mix_frequency: float*, *weights: ndarray*)

Bases: object

Configuration for a narrowband stream.

decimation: int

Factor by which bandwidth is reduced

mix_frequency: float

Mixer frequency, in cycles per ADC sample

weights: ndarray

Downconversion filter weights (float)

katgpucbf.fgpu.ddc module

Digital down-conversion.

class katgpucbf.fgpu.ddc.**DDC**(*template:* DDCTemplate, *command_queue: AbstractCommandQueue*, *samples: int*, *n_pols: int*)

Bases: Operation

Operation implementating *DDCTemplate*.

The kernel takes 10-bit integer inputs (real) and produces 32-bit floating-point outputs (complex). The user provides a real-valued FIR baseband filter and a mixer frequency for translating the signal from the desired band to baseband.

Element j of the output contains the dot product of **weights** with elements $dj, d(j+1), \ldots, d(j+taps-1)$ of the mixed signal. The mixed signal is the product of sample j of the input with $e^{2\pi i(aj+b)}$, where a is the *mix_frequency* argument to *configure()* and b is the (settable) `mix_phase` property.

Slots

in

[samples * input_sample_bits // BYTE_BITS, uint8] Input digitiser samples in a big chunk.

out

[out_samples, complex64] Filtered and subsampled output data

Raises

ValueError – If *samples* is not a multiple of 8, or is less than *template.taps*

Parameters

- **template** – Template for the PFB-FIR operation.
- **command_queue** – The GPU command queue
- **samples** – Number of input samples to store, per polarisation
- **n_pols** – Number of polarisations

configure(*mix_frequency: float*, *weights: ndarray*) → None

Set the mixer frequency and filter weights.

This is a somewhat expensive operation, as it computes lookup tables and transfers them to the device synchronously. It is only intended to be used at startup rather than continuously.

Note: The provided *mix_frequency* is quantised. The actual mixer frequency can be retrieved from the *mix_frequency* property.

property **mix_frequency**: float

Mixer frequency in cycles per ADC sample.

class katgpucbf.fgpu.ddc.**DDCTemplate**(*context: AbstractContext*, *taps: int*, *subsampling: int*, *input_sample_bits: int*, *tuning: _TuningDict | None = None*)

Bases: object

Template for digital down-conversion.

See *DDC* for a more detailed description of what it does.

Parameters

- **context** – The GPU context that we'll operate in
- **taps** – Number of taps in the FIR filter
- **subsampling** – Fraction of samples to retain after filtering
- **input_sample_bits** – Bits per input sample

classmethod **autotune**(*context: AbstractContext*, *taps: int*, *subsampling: int*, *input_sample_bits: int*) → _TuningDict

Determine tuning parameters.

autotune_version = 2

instantiate(*command_queue: AbstractCommandQueue*, *samples: int*, *n_pols: int*) → *DDC*

Generate a *DDC* object based on the template.

static **unroll_align**(*subsampling: int*, *input_sample_bits: int*) → int

Determine the factor that must divide into *unroll*.

katgpucbf.fgpu.delay module

A collection of classes and methods for delay-tracking.

It should be noted that the classes in this module use a slightly different model than the public katcp interface. The reference channel for phase change is channel 0, rather than the centre channel. The difference is dealt with by the request handler for the `?delays` katcp request.

class katgpucbf.fgpu.delay.**AbstractDelayModel**

Bases: ABC

Abstract base class for delay models.

All units are samples rather than SI units.

abstract **range**(*start: int*, *stop: int*, *step: int*) → tuple[ndarray, ndarray, ndarray]

Find input timestamps corresponding to a range of output samples.

For each output sample with timestamp in `range(start, stop, step)`, it determines a corresponding input sample.

Parameters

- **start** – First timestamp (inclusive).
- **stop** – Last timestamp (exclusive)
- **step** – Interval between timestamps (must be positive).

Returns

- *orig_time* – Undelayed integer timestamps corresponding to `range(start, stop, step)`
- *residual* – Fractional sample delay not accounted for by `time - orig_time`.
- *phase* – Fringe-stopping phase to be added.

abstract skip(*target: int*, *start: int*, *step: int*) → int

Find the next output time for which the input time is at least *target*.

The output time must also be at least *start* and a multiple of *step*.

class katgpucbf.fgpu.delay.**AlignedDelayModel**(*base: _DM*, *align: int*)

Bases: *AbstractDelayModel*, Generic[_DM]

Wrap another delay model and enforce an alignment on original timestamp.

Note that this can cause residual delays to be larger than 1.

range(*start: int*, *stop: int*, *step: int*) → tuple[ndarray, ndarray, ndarray]

Find input timestamps corresponding to a range of output samples.

For each output sample with timestamp in `range(start, stop, step)`, it determines a corresponding input sample.

Parameters

- **start** – First timestamp (inclusive).
- **stop** – Last timestamp (exclusive)
- **step** – Interval between timestamps (must be positive).

Returns

- *orig_time* – Undelayed integer timestamps corresponding to `range(start, stop, step)`
- *residual* – Fractional sample delay not accounted for by `time - orig_time`.
- *phase* – Fringe-stopping phase to be added.

skip(*target: int*, *start: int*, *step: int*) → int

Find the next output time for which the input time is at least *target*.

The output time must also be at least *start* and a multiple of *step*.

class katgpucbf.fgpu.delay.**LinearDelayModel**(*start: int*, *delay: float*, *delay_rate: float*, *phase: float*, *phase_rate: float*)

Bases: *AbstractDelayModel*

Delay model that adjusts delay linearly over time.

Parameters

- **start** – Output sample at which the model should start being used.
- **delay** – Delay to apply at *start*. [seconds]
- **delay_rate** – Rate of change of delay. [seconds/second]
- **phase** – Fringe-stopping phase to apply with the fine delay. [radians]
- **phase_rate** – Rate of change of the fringe-stopping phase. [radians/second]

Raises

ValueError – if *rate* is greater than or equal to 1 or *start* is negative

range(*start: int*, *stop: int*, *step: int*) → tuple[ndarray, ndarray, ndarray]

Find input timestamps corresponding to a range of output samples.

For each output sample with timestamp in `range(start, stop, step)`, it determines a corresponding input sample.

Parameters

- **start** – First timestamp (inclusive).
- **stop** – Last timestamp (exclusive)
- **step** – Interval between timestamps (must be positive).

Returns

- *orig_time* – Undelayed integer timestamps corresponding to `range(start, stop, step)`
- *residual* – Fractional sample delay not accounted for by `time - orig_time`.
- *phase* – Fringe-stopping phase to be added.

skip(*target: int*, *start: int*, *step: int*) → int

Find the next output time for which the input time is at least *target*.

The output time must also be at least *start* and a multiple of *step*.

class katgpucbf.fgpu.delay.**MultiDelayModel**(*callback_func: Callable[[Sequence[LinearDelayModel]], None] | None = None*)

Bases: *AbstractDelayModel*

Piece-wise linear delay model.

The model evolves over time by calling *add()*. It **must** only be queried with monotonically increasing *start* values, because as soon as a query is made beyond the end of the first piece it is discarded. Additionally, after calling *skip()*, the return value should be treated as a lower bound for future *start* values.

In the initial state it has a model with zero delay.

It accepts an optional callback function that takes in the LinearDelayModels attached to this MultiDelayModel. This callback is called whenever the first linear piece changes. It is also called immediately by the constructor.

add(*model: LinearDelayModel*) → None

Extend the model with a new linear model.

The new model is applicable from its start time forever. If the new model has an earlier start time than some previous model, the previous model will be discarded.

range(*start: int*, *stop: int*, *step: int*) → tuple[ndarray, ndarray, ndarray]

Find input timestamps corresponding to a range of output samples.

For each output sample with timestamp in `range(start, stop, step)`, it determines a corresponding input sample.

Parameters

- **start** – First timestamp (inclusive).
- **stop** – Last timestamp (exclusive)
- **step** – Interval between timestamps (must be positive).

Returns

- *orig_time* – Undelayed integer timestamps corresponding to `range(start, stop, step)`
- *residual* – Fractional sample delay not accounted for by `time - orig_time`.
- *phase* – Fringe-stopping phase to be added.

skip(*target: int*, *start: int*, *step: int*) → int

Find the next output time for which the input time is at least *target*.

The output time must also be at least *start* and a multiple of *step*.

exception katgpucbf.fgpu.delay.**NonMonotonicQueryWarning**

Bases: UserWarning

Delay model was queried non-monotonically.

katgpucbf.fgpu.delay.**wrap_angle**(*angle*)

Restrict an angle to [-pi, pi].

This works on both Python scalars and numpy arrays.

katgpucbf.fgpu.engine module

katgpucbf.fgpu.main module

katgpucbf.fgpu.output module

Data structures capturing static configuration of a single output stream.

class katgpucbf.fgpu.output.**NarrowbandOutput**(*name: str*, *channels: int*, *jones_per_batch: int*, *taps: int*, *w_cutoff: float*, *dst: list[Endpoint]*, *centre_frequency: float*, *decimation: int*, *ddc_taps: int*, *weight_pass: float*)

Bases: *Output*

Static configuration for a narrowband stream.

centre_frequency: float

ddc_taps: int

decimation: int

property internal_channels: int

Number of channels in the PFB.

property internal_decimation: int

Factor by which bandwidth is reduced by the DDC kernel.

property spectra_samples: int

Number of incoming digitiser samples needed per spectrum.

Note that this is the spacing between spectra. Each spectrum uses an overlapping window with more samples than this.

property subsampling: int

Number of digitiser samples between PFB input samples.

weight_pass: *float*

property window: *int*

Number of digitiser samples that contribute to each output spectrum.

class katgpucbf.fgpu.output.**Output**(*name: str*, *channels: int*, *jones_per_batch: int*, *taps: int*, *w_cutoff: float*, *dst: list[Endpoint]*)

Bases: ABC

Static configuration for an output stream.

channels: *int*

dst: *list[Endpoint]*

abstract property internal_channels: *int*

Number of channels in the PFB.

abstract property internal_decimation: *int*

Factor by which bandwidth is reduced by the DDC kernel.

jones_per_batch: *int*

name: *str*

property spectra_per_heap: *int*

Number of spectra in each output heap.

abstract property spectra_samples: *int*

Number of incoming digitiser samples needed per spectrum.

Note that this is the spacing between spectra. Each spectrum uses an overlapping window with more samples than this.

abstract property subsampling: *int*

Number of digitiser samples between PFB input samples.

taps: *int*

w_cutoff: *float*

abstract property window: *int*

Number of digitiser samples that contribute to each output spectrum.

class katgpucbf.fgpu.output.**WidebandOutput**(*name: str*, *channels: int*, *jones_per_batch: int*, *taps: int*, *w_cutoff: float*, *dst: list[Endpoint]*)

Bases: *Output*

Static configuration for a wideband output stream.

property decimation: *int*

property internal_channels: *int*

Number of channels in the PFB.

property internal_decimation: *int*

Factor by which bandwidth is reduced by the DDC kernel.

property spectra_samples: int

Number of incoming digitiser samples needed per spectrum.

Note that this is the spacing between spectra. Each spectrum uses an overlapping window with more samples than this.

property subsampling: int

Number of digitiser samples between PFB input samples.

property window: int

Number of digitiser samples that contribute to each output spectrum.

katgpucbf.fgpu.pfb module

PFB module.

These classes handle the operation of the GPU in performing the PFB-FIR part through a mako-templated kernel.

class katgpucbf.fgpu.pfb.**PFBFIR**(*template:* PFBFIRTemplate, *command_queue: AbstractCommandQueue*, *samples: int*, *spectra: int*)

Bases: Operation

The windowing FIR filters that form the first part of the PFB.

The best place to look in order to understand how these work from a strictly DSP sense is Danny C. Price's paper [Pri].

In general the operation can read some interval of the input slot and write to some interval of the output slot. The sizes of these slots need not be related. This can be useful to build up a larger output from smaller invocations that have different coarse delays.

Slots

The slots depend on `template.complex_input`. If it is false, then they are

in
[pols × (samples * input_sample_bits // BYTE_BITS), uint8] Input samples in a big chunk.

out
[pols × spectra × 2*channels, float32] FIR-filtered time data, ready to be processed by the FFT.

weights
[2*channels*taps, float32] The time-domain transfer function of the FIR filter to be applied.

total_power
[pols, uint64] Sum of squares of input samples. This will not include every input sample. Rather, it will contain a specific tap from each PFB window (currently, the last tap, but that is an implementation detail).

This is incremented rather than overwritten. It is the caller's responsibility to zero it when desired, or alternatively to track values before and after to measure the change.

Otherwise, they are

in
[samples, complex64] Input samples

out
[spectra × channels, complex64] See above

weights
[channels*taps, float32] See above

Raises

- **ValueError** – If `samples` is not a multiple of 8 and `complex_input` is false
- **ValueError** – If `samples` is too large (more than 2**29)

Parameters

- **template** – Template for the PFB-FIR operation.
- **command_queue** – The GPU command queue (typically this will be an instance of `katsdpsigproc.cuda.CommandQueue` which wraps a CUDA Stream) on which actual processing operations are to be scheduled.
- **samples** – Number of input samples that will be processed each time the operation is run.
- **spectra** – Number of spectra that we will get from each chunk of samples.

class katgpucbf.fgpu.pfb.**PFBFIRTemplate**(*context: AbstractContext*, *taps: int*, *channels: int*, *input_sample_bits: int*, *unzip_factor: int = 1*, ***, *complex_input: bool = False*, *n_pols: int*)

Bases: `object`

Template for the PFB-FIR operation.

The operation can operate in two different modes. In the first mode (intended for a wideband channeliser), the input contains real digitiser samples (bit-packed integers). In the second mode (intended for a narrowband channeliser), the digitiser samples have already been preprocessed and the PFB operates on complex-valued inputs (floating point). The mode is selected with the *complex_input* parameter.

Parameters

- **context** – The GPU context that we'll operate in.
- **taps** – The number of taps that you want the resulting PFB-FIRs to have.
- **channels** – Number of channels into which the input data will be decomposed.
- **input_sample_bits** – Bits per each component of input. If *complex_input* is true, the input values are floating-point complex numbers and this must equal 32. Otherwise, the inputs are packed integers, and the value must be in `DIG_SAMPLE_BITS_VALID`.
- **unzip_factor** – The output is reordered so that every unzip_factor'ith pair of outputs (or single complex output, if *complex_input* is true) is placed contiguously.
- **complex_input** – Operation mode (see above).
- **n_pols** – Number of polarisations to operate over. The polarisations are stored contiguously in memory, but have independent offsets.

Raises

- **ValueError** – If `taps` is not positive.
- **ValueError** – If `complex_input` is true and `input_sample_bits` is not 32.
- **ValueError** – If `complex_input` is false and `input_sample_bits` is not in *DIG_SAMPLE_BITS_VALID*.
- **ValueError** – If `channels` is not an even power of 2.
- **ValueError** – If `channels` is not a multiple of `unzip_factor`.

- **ValueError** – If `2*channels` is not a multiple of the workgroup size (currently 128).

instantiate(*command_queue: AbstractCommandQueue*, *samples: int*, *spectra: int*) → *PFBFIR*

Generate a *PFBFIR* object based on the template.

katgpucbf.fgpu.postproc module

Postproc module.

These classes handle the operation of the GPU in performing the fine-delay, per-channel gains, requantisation and corner-turn through a mako-templated kernel.

class katgpucbf.fgpu.postproc.**Postproc**(*template: PostprocTemplate*, *command_queue: AbstractCommandQueue*, *spectra: int*, *spectra_per_heap: int*)

Bases: `Operation`

The fine-delay, requant and corner-turn operations coming after the PFB.

Slots

in
[N_POLS × spectra × unzip_factor × channels // unzip_factor, complex64] Input channelised data for the two polarisations. These are formed by taking the complex-to-complex Fourier transform of the input reinterpreted as a complex input. See *FFT* for details.

out
[spectra // spectra_per_heap × out_channels × spectra_per_heap × N_POLS] Output F-engine data, quantised and corner-turned, ready for transmission on the network. See *gaussian_dtype()* for the type.

saturated
[spectra // spectra_per_heap × N_POLS, uint32] Number of saturated complex values in **out**.

fine_delay
[spectra × N_POLS, float32] Fine delay in samples (one value per pol).

phase
[spectra × N_POLS, float32] Fixed phase adjustment in radians (one value per pol).

gains
[out_channels × N_POLS, complex64] Per-channel gain (one value per pol).

Parameters

- **template** (`PostprocTemplate`) – The template for the post-processing operation.
- **command_queue** (*AbstractCommandQueue*) – The GPU command queue (typically this will be a CUDA Stream) on which actual processing operations are to be scheduled.
- **spectra** (*int*) – Number of spectra on which post-prodessing will be performed.
- **spectra_per_heap** (*int*) – Number of spectra to send out per heap.

class katgpucbf.fgpu.postproc.**PostprocTemplate**(*context: AbstractContext*, *channels: int*, *unzip_factor: int = 1*, ***, *complex_pfb: bool*, *out_bits: int*, *out_channels: tuple[int, int] | None = None*)

Bases: `object`

Template for the postproc operation.

Parameters

- **context** – The GPU context that we'll operate in.
- **channels** – Number of input channels in each spectrum.
- **unzip_factor** – Radix of the final Cooley-Tukey FFT step performed by the kernel.
- **complex_pfb** – If true, the PFB is a complex-to-complex transform, and no real-to-complex fixup is needed. Additionally, the DC channel is considered to be the centre of the band i.e. it is written to the middle of the output rather than the start (and similarly, gains for it are loaded from the middle of the gain array etc).
- **out_bits** – Bits per real/imaginary value. Only 4 or 8 are currently supported. When 4, the real part is in the most-significant bits.
- **out_channels** – Range of channels to write to the output (defaults to all).

instantiate(*command_queue: AbstractCommandQueue*, *spectra: int*, *spectra_per_heap: int*) → *Postproc*

Generate a *Postproc* object based on this template.

katgpucbf.fgpu.recv module

Recv module.

class katgpucbf.fgpu.recv.**Chunk**(*self: spead2._spead2.recv.Chunk*, ***kwargs*)

Bases: Chunk

Collection of heaps passed to the GPU at one time.

It extends the spead2 base class to store a timestamp (computed from the chunk ID when the chunk is received), and optionally store a vkgdr device array.

When used as a context manager, it will call *recycle()* on exit.

device: object

recycle() → None

Return the chunk to the owning stream/group.

sink: ReferenceType

timestamp: int

class katgpucbf.fgpu.recv.**Layout**(*sample_bits: int*, *heap_samples: int*, *chunk_samples: int*, *mask_timestamp: bool*)

Bases: *BaseLayout*

Parameters controlling the sizes of heaps and chunks.

property chunk_heaps: int

Number of heaps per chunk, on time axis.

chunk_samples: int

property heap_bytes: int

Number of payload bytes per heap.

heap_samples: int

mask_timestamp: bool

sample_bits: int

property timestamp_mask: uint64

Mask to AND with incoming timestamps.

async katgpucbf.fgpu.recv.**iter_chunks**(*ringbuffer:* *ChunkRingbuffer*, *layout:* Layout, *sensors:* *SensorSet*, *time_converter:* TimeConverter) → AsyncGenerator[*Chunk*, None]

Iterate over the chunks and update sensors.

It also populates the chunk timestamp.

Parameters

- **ringbuffer** – Source of chunks.
- **layout** – Structure of the streams.
- **sensors** – Sensor set containing at least the sensors created by `make_sensors()`.
- **time_converter** – Converter to turn data timestamps into sensor timestamps.

katgpucbf.fgpu.send module

Network transmission handling.

class katgpucbf.fgpu.send.**Chunk**(*data:* *ndarray*, *saturated:* *ndarray*, *, *n_substreams:* *int*, *feng_id:* *int*, *spectra_samples:* *int*)

Bases: object

An array of frames, spanning multiple timestamps.

Parameters

- **data** – Storage for voltage data, with shape (n_frames, n_channels, n_spectra_per_heap, N_POLS) and a dtype returned by *gaussian_dtype()*.
- **saturated** – Storage for saturation counts, with shape (n_frames, N_POLS) and dtype uint32.
- **n_substreams** – Number of substreams over which the data will be divided (must divide evenly into the number of channels).
- **feng_id** – F-Engine ID to place in the SPEAD heaps
- **spectra_samples** – Difference in timestamps between successive frames

cleanup: Callable[[], None] | None

Callback to return the chunk to the appropriate queue

present

Whether each frame has valid data

async **send**(*streams:* *list[spead2.send.asyncio.AsyncStream]*, *frames:* *int*, *time_converter:* TimeConverter, *sensors:* *SensorSet*, *output_name:* *str*) → None

Transmit heaps over SPEAD streams.

Frames from 0 to *frames* - 1 are sent asynchronously. The contents of each frame are distributed over the streams. If the number of streams does not divide into the number of destination endpoints, there will be imbalances, because the partitioning is the same for every frame.

property timestamp: int

Timestamp of the first heap.

Setting this property updates the timestamps stored in all the heaps. This should not be done while a previous call to *send()* is still in progress.

class katgpucbf.fgpu.send.**Frame**(*timestamp: ndarray*, *data: ndarray*, *saturated: ndarray*, *, *n_substreams: int*, *feng_id: int*)

Bases: object

Holds all the heaps for a single timestamp.

It does not own its memory - the backing store is in *Chunk*.

Parameters

- **timestamp** – Zero-dimensional array of dtype >u8 holding the timestamp.
- **data** – Payload data for the frame, of shape (channels, spectra_per_heap, N_POLS).
- **saturated** – Saturation data for the frame, of shape (N_POLS,)
- **feng_id** – Value to put in feng_id SPEAD item
- **n_substreams** – Number of substreams into which the channels are divided

katgpucbf.fgpu.send.**PREAMBLE_SIZE = 72**

Number of non-payload bytes per packet (header, 8 items pointers)

katgpucbf.fgpu.send.**make_descriptor_heap**(*, *channels_per_substream: int*, *spectra_per_heap: int*, *sample_bits: int*) → Heap

Create a descriptor heap for output F-Engine data.

katgpucbf.fgpu.send.**make_streams**(*, *output_name: str*, *thread_pool: ThreadPool*, *endpoints: list[Endpoint]*, *interfaces: list[str]*, *ttl: int*, *ibv: bool*, *packet_payload: int*, *comp_vector: int*, *buffer: int*, *bandwidth: float*, *send_rate_factor: float*, *feng_id: int*, *num_ants: int*, *n_data_heaps: int*, *chunks: Sequence[*Chunk*]*) → list[spead2.send.asyncio.AsyncStream]

Create asynchronous SPEAD streams for transmission.

Each stream is configured with substreams for all the end-points. They differ only in the network interface used (there is one per interface). Thus, they can be used interchangeably for load-balancing purposes.

Module contents

katgpucbf.fgpu.**DIG_SAMPLE_BITS_VALID = [2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 16]**

Valid values for the --dig-sample-bits command-line option

20.1.3 katgpucbf.fsim package

Submodules

katgpucbf.fsim.main module

Simulate channelised data from the MeerKAT F-Engines destined for one or more XB-Engines.

Refer to *F-Engine Packet Simulator* for more information.

katgpucbf.fsim.main.**QUEUE_DEPTH = 8**

Number of heaps in time to keep in flight

class katgpucbf.fsim.main.**Sender**(*args: Namespace*, *idx: int*)

Bases: object

Manage sending data to a single XB-engine.

async run(*sync_time: float*, *run_once: bool*) → None

Send heaps until cancelled.

async katgpucbf.fsim.main.**async_main**() → None

Run main program.

katgpucbf.fsim.main.**main**() → None

Run main program.

katgpucbf.fsim.main.**make_heap**(*timestamp: ndarray*, *feng_id: int*, *channel_offset: int*, *payload: ndarray*) → HeapReference

Create a heap to transmit.

The *timestamp* must be a zero-dimensional array of dtype >u8. It will be transmitted by reference, so it can be updated in place to change the stored timestamp.

katgpucbf.fsim.main.**make_heap_payload**(*out: ndarray*, *heap_index: int*, *feng_id: int*, *n_ants: int*) → None

Create the simulated payload data for a heap.

A pattern is chosen that will hopefully be easy to verify at the receiver graphically. On each F-Engine, the signal amplitude will increase linearly over time for each channel. Each channel will have a different starting amplitude but the rate of increase will be the same for all channels.

Each F-Engine will have the same same signal amplitude for the same timestamp, but the signal phase will be different. The signal phase remains constant across all channels in a single F-Engine. By examining the signal phase it can be verified that correct feng_id is attached to the correct data.

These samples need to be stored as 8 bit samples. As such, the amplitude is wrapped each time it reaches 127. 127 is used as the amplitude when multiplied by the phase can reach -127. The full range of values is covered.

This current format is not fixed and it is likely that it will be adjusted to be suited for different verification needs.

Parameters

- **out** – Output array, with shape (n_channels_per_substream, n_spectra_per_heap, N_POLS, COMPLEX)
- **heap_index** – Heap index on time axis
- **feng_id** – Heap index on antenna axis
- **n_ants** – Number of antennas in the array

katgpucbf.fsim.main.**make_stream**(*args: Namespace*, *idx: int*, *data: ndarray*) → spead2.send.asyncio.AsyncStream

Create a SPEAD stream for a single destination.

Parameters

- **args** – Command-line arguments
- **idx** – Index into the destinations to use
- **data** – All payload data for this destination

katgpucbf.fsim.main.**parse_args**(*arglist: Sequence[str] | None = None*) → Namespace

Parse the command-line arguments.

Module contents

20.1.4 katgpucbf.xbgpu package

Submodules

katgpucbf.xbgpu.beamform module

Implement the calculations for beamforming.

class katgpucbf.xbgpu.beamform.**Beamform**(*template: BeamformTemplate*, *command_queue: AbstractCommandQueue*, *n_frames: int*, *n_ants: int*, *n_channels: int*, *seed: int*, *sequence_first: int*, *sequence_step: int = 1*)

Bases: Operation

Operation for beamforming.

For ease-of-use with the data formats used in the rest of katgpucbf, time is split into two dimensions: a coarse outer dimension (called "frames") and a finer inner dimension ("spectra").

Slots

in
: [n_frames × n_ants × n_channels × n_spectra_per_frame × N_POLS × COMPLEX, int8] Complex (Gaussian integer) input channelised voltages

out
: [n_frames × n_beams × n_channels × n_spectra_per_frame × COMPLEX, int8] Complex (Gaussian integer) output channelised voltages

saturated: n_beams, uint32
: Number of saturated output values, per beam. This value is *incremented* by the kernel, so should be explicitly zeroed first if desired.

weights
: [n_ants × n_beams, complex64] Complex scale factor to apply to each antenna for each beam

delays
: [n_ants × n_beams, float32] Delay used to compute channel-dependent phase rotation. The rotation applied is $e^{j\pi cd}$ where c is the channel number and d is the delay value. Note that this will not apply any rotation to the first channel in the data; any such rotation needs to be baked into **weights**.

rand_states
: [n_frames × n_channels × n_spectra_per_frame, curandStateXORWOW_t (packed)] Independent random states for generating dither values. This is set up by the constructor and should not normally need to be touched.

Parameters

- **template** – The template for the operation
- **command_queue** – The command queue on which to enqueue the work

- **n_frames** – Number of frames (coarse time dimension)
- **n_ants** – Number of antennas
- **n_channels** – Number of frequency channels
- **seed** – See *RandomStateBuilder*.
- **sequence_first** – See *RandomStateBuilder*.
- **sequence_step** – See *RandomStateBuilder*.

class katgpucbf.xbgpu.beamform.**BeamformTemplate**(*context: AbstractContext*, *beam_pols: Sequence[int]*, *n_spectra_per_frame: int*)

Bases: object

Template for beamforming.

Parameters

- **context** – The GPU context that we'll operate in.
- **beam_pols** – One entry per single-polarisation output beam. Each entry is either 0 or 1, to indicate which input polarisation to use in the beam.
- **n_spectra_per_frame** – Number of samples in time axis for each frame (fine time dimension) - see *Beamform*.

instantiate(*command_queue: AbstractCommandQueue*, *n_frames: int*, *n_ants: int*, *n_channels: int*, *seed: int*, *sequence_first: int*, *sequence_step: int = 1*) → *Beamform*

Generate a *Beamform* object based on the template.

katgpucbf.xbgpu.bsend module

Module for sending tied array channelised voltage products onto the network.

class katgpucbf.xbgpu.bsend.**BSend**(*outputs: Sequence[*BOutput*]*, *frames_per_chunk: int*, *n_tx_items: int*, *n_channels: int*, *n_channels_per_substream: int*, *spectra_per_heap: int*, *adc_sample_rate: float*, *timestamp_step: int*, *send_rate_factor: float*, *channel_offset: int*, *context: AbstractContext*, *stream_factory: Callable[[StreamConfig, Sequence[ndarray]], spead2.send.asyncio.AsyncStream]*, *packet_payload: int = 8192*, *tx_enabled: bool = False*)

Bases: object

Class for turning tied array channelised voltage products into SPEAD heaps.

This class creates a queue of chunks that can be sent out onto the network. To obtain a chunk, call *get_free_chunk()* - which will return a *Chunk*. This object will create a limited number of transmit buffers and keep recycling them, avoiding any memory allocation at runtime.

The transmission of a chunk's data is abstracted by *send_chunk()*. This invokes transmission and immediately returns the *Chunk* back to the queue for reuse.

This object keeps track of each tied-array-channelised-voltage data stream by means of a substreams in spead2.send.asyncio.AsyncStream, allowing for individual enabling and disabling of the data product.

To allow this class to be used with multiple transports, the constructor takes a factory function to create the stream.

Parameters

- **outputs** – Sequence of *output.BOutput*.
- **frames_per_chunk** – Number of *Frame*s in each transmitted *Chunk*.
- **n_tx_items** – Number of *Chunk* to create.
- **adc_sample_rate** – See `XBEngine` for further information.
- **n_channels** – See `XBEngine` for further information.
- **n_channels_per_substream** – See `XBEngine` for further information.
- **spectra_per_heap** – See `XBEngine` for further information.
- **channel_offset** – See `XBEngine` for further information.
- **timestamp_step** – The timestamp step between successive heaps.
- **send_rate_factor** – Factor dictating how fast the send-stream should transmit data.
- **context** – Device context to create buffers.
- **stream_factory** – Callback function to create the spead2 send stream. It is passed the stream configuration and memory buffers.
- **packet_payload** – Size, in bytes, for the output packets (tied array channelised voltage payload only; headers and padding are added to this).
- **tx_enabled** – Enable/Disable transmission.

descriptor_heap: Heap

enable_substream(*stream_id: int*, *enable: bool = True*) → None

Enable/Disable a substream's data transmission.

BSend operates as a large single stream with multiple substreams. Each substream is its own data product and is required to be enabled/disabled independently.

Parameters

- **stream_id** – Index of the substream's data product.
- **enable** – Boolean indicating whether the *stream_id* should be enabled or disabled.

async get_free_chunk() → *Chunk*

Obtain a *Chunk* for transmission.

We await the chunk's `future` to be sure we are not overwriting data that is still being transmitted. If sending failed, it is no longer being transmitted, and therefore safe to return the chunk.

Raises

asyncio.CancelledError – If the chunk's send future is cancelled.

header_size: Final[int] = 64

send_chunk(*chunk:* Chunk, *time_converter:* TimeConverter, *sensors: SensorSet*) → None

Send a chunk's data and put it on the `_chunks_queue`.

async send_stop_heap() → None

Send a Stop Heap over the spead2 transport.

class katgpucbf.xbgpu.bsend.**Chunk**(*data: ndarray*, *saturated: ndarray*, *, *channel_offset: int*, *timestamp_step: int*)

Bases: `object`

An array of *Frame*s.

Parameters

- **data** – Storage for tied-array-channelised-voltage data, with shape (n_frames, n_beams, n_channels_per_substream, n_spectra_per_heap, COMPLEX) and dtype SEND_DTYPE.
- **saturated** – Storage for saturation counts, with shape (n_beams,) and dtype uint32.
- **channel_offset** – The first frequency channel processed.
- **timestamp_step** – Timestamp step between successive *Frame*s in a chunk.

property present_ants: ndarray

Number of antennas present in the current beam sums.

This is a count for each *Frame* in the chunk. Setting this property updates the immediate SPEAD items in the heaps. Much like *timestamp*, this should only be done when future is done.

send(*send_stream:* BSend, *time_converter:* TimeConverter, *sensors:* *SensorSet*) → Future

Transmit a chunk's heaps over a SPEAD stream.

This method returns immediately and sends the data asynchronously. Before modifying the chunk, first await future.

property timestamp: int

Timestamp of the first heap.

Setting this property updates the timestamps stored in all the heaps. This should only be done when future is done.

class katgpucbf.xbgpu.bsend.**Frame**(*timestamp:* *ndarray*, *data:* *ndarray*, *, *channel_offset:* *int*, *present_ants:* *ndarray*)

Bases: object

Hold all data for heaps with a single timestamp.

It does not own its memory - the backing store is in *Chunk*. It keeps a cached spead2.send.HeapReferenceList with the heaps of the enabled beams, along with a version counter that is used to invalidate it.

Parameters

- **timestamp** – Zero-dimensional array of dtype >u8 holding the timestamp
- **data** – Payload data for the frame with shape (n_beams, n_channels_per_substream, spectra_per_heap, COMPLEX).
- **channel_offset** – The first frequency channel processed.
- **present_ants** – Zero-dimensional array of dtype >u8 holding the number of antennas present in the Frame's input data.

katgpucbf.xbgpu.bsend.**make_stream**(*, *output_names:* *list[str]*, *endpoints:* *list[Endpoint]*, *interface:* *str*, *ttl:* *int*, *use_ibv:* *bool*, *affinity:* *int*, *comp_vector:* *int*, *stream_config:* *StreamConfig*, *buffers:* *Sequence[ndarray]*) → spead2.send.asyncio.AsyncStream

Create asynchronous SPEAD stream for transmission.

This is architected to be a single send stream with multiple substreams, each corresponding to a tied-array-channelised-voltage output data product. The *endpoints* need not be a contiguous list of multicast addresses.

katgpucbf.xbgpu.correlation module

Module wrapping the ASTRON Tensor-Core Correlation Kernels in the MeerKAT katsdpsigproc framework.

Todo: Eventually modify the classes to support 4 and 16 bit input samples. The kernel supports this, but it is not exposed to the reader. There is no use case for this at the moment, so this is a low priority.

class katgpucbf.xbgpu.correlation.**Correlation**(*template:* CorrelationTemplate, *command_queue:* *AbstractCommandQueue*, *n_batches:* *int*)

Bases: `Operation`

Tensor-Core correlation kernel.

Specifies the shape of the input sample and output visibility buffers required by the kernel. The parameters specified in the *CorrelationTemplate* object are used to determine the shape of the buffers. There is an outermost dimension called "batches", over which the operation is parallelised. Not all batches need to be processed every time: set the `first_batch` and `last_batch` attributes to control which batches will be computed.

The input sample buffer must have the shape: `[n_batches][n_ants][channels][spectra_per_heap][polarisations]`

There is an alignment requirement for `spectra_per_heap` due to the implementation details of the kernel. For 8-bit input mode, `spectra_per_heap` must be a multiple of 16.

Each input element is a complex 8-bit integer sample. `numpy` does not support 8-bit complex numbers, so the dimensionality is extended by 1, with the last dimension sized 2 to represent the complexity.

With 8-bit input samples, the value -128i is not supported by the kernel as there is no 8-bit complex conjugate representation of this number. Passing `-128i` into the kernel will produce incorrect values at the output.

The output visibility buffer must have the shape `[channels][baselines][COMPLEX]`. In 8-bit mode, each element in this visibility matrix is a 32-bit integer value.

Calling this object does not directly update the output. Instead, it updates an intermediate buffer (called `mid_visibilities`). To produce the output, call *reduce()*. This function can also flag data that was missing during the accumulation, by writing a special value. This is controlled by the `present_baselines` slot, which has one boolean entry per baseline (antenna pair).

Currently only 8-bit input mode is supported.

static get_baseline_index(*ant1:* *int*, *ant2:* *int*) → int

Get index in the visibilities matrix for baseline (ant1, ant2).

The visibilities matrix indexing is as follows:

```
     ant2 = 0  1  2  3  4
          +---------------
ant1 = 0 | 00 01 03 06 10
       1 |    02 04 07 11
       2 |       05 08 12
       3 |          09 13
       4 |             14
```

This function requires that $ant2 \geq ant1$

reduce() → None

Finalise computation of the output visibilities from the internal buffer.

zero_visibilities() → None

Zero all the values in the internal buffer.

class katgpucbf.xbgpu.correlation.**CorrelationTemplate**(*context: AbstractContext*, *n_ants: int*, *n_channels: int*, *n_spectra_per_heap: int*, *input_sample_bits: int*)

Bases: object

Template class for the Tensor-Core correlation kernel.

The template creates a *Correlation* that will run the compiled kernel. The parameters are used to compile the kernel and by the *Correlation* to specify the shape of the memory buffers connected to this kernel.

The number of baselines calculated here is not the canonical way that it is done in radio astronomy:

$$n_{baselines} = \frac{n_{ants} * (n_{ants} + 1)}{2}$$

Because we have a dual-polarised telescope, we calculate four 'baselines' for each canonical baseline as calculated above, namely h_1h_2, h_1v_2, v_1h_2, and v_1v_2. So the list of baselines appears four times as long as you might expect.

Parameters

- **n_ants** – The number of antennas that will be correlated. Each antennas is expected to produce two polarisations.
- **n_channels** – The number of frequency channels to be processed.
- **n_spectra_per_heap** – The number of time samples to be processed per frequency channel.
- **input_sample_bits** – The number of bits per input sample. Only 8 bits is supported at the moment.

instantiate(*command_queue: AbstractCommandQueue*, *n_batches: int*) → *Correlation*

Create a *Correlation* using this template to build the kernel.

katgpucbf.xbgpu.correlation.**MIN_COMPUTE_CAPABILITY = (7, 2)**

Minimum CUDA compute capability needed for the kernel (with 8-bit samples)

katgpucbf.xbgpu.correlation.**MISSING = array([-2147483648, 1], dtype=int32)**

Magic value indicating missing data

katgpucbf.xbgpu.correlation.**device_filter**(*device: AbstractDevice*) → bool

Determine whether a device is suitable for running the kernel.

katgpucbf.xbgpu.engine module

katgpucbf.xbgpu.main module

katgpucbf.xbgpu.output module

Data structures capturing static configuration of a single output stream.

class katgpucbf.xbgpu.output.**BOutput**(*name: str*, *dst: Endpoint*, *pol: int*)

Bases: *Output*

Static configuration for an output beam stream.

pol: *int*

class katgpucbf.xbgpu.output.**Output**(*name: str*, *dst: Endpoint*)

Bases: ABC

Static configuration for an output stream.

dst: *Endpoint*

name: *str*

class katgpucbf.xbgpu.output.**XOutput**(*name: str*, *dst: Endpoint*, *heap_accumulation_threshold: int*)

Bases: *Output*

Static configuration for an output baseline-correlation-products stream.

heap_accumulation_threshold: *int*

katgpucbf.xbgpu.recv module

SPEAD receiver utilities.

class katgpucbf.xbgpu.recv.**Layout**(*n_ants: int*, *n_channels_per_substream: int*, *n_spectra_per_heap: int*, *timestamp_step: int*, *sample_bits: int*, *heaps_per_fengine_per_chunk: int*)

Bases: *BaseLayout*

Parameters controlling the sizes of heaps and chunks.

Parameters

- **n_ants** (*int*) – The number of antennas that data will be received from
- **n_channels_per_substream** (*int*) – The number of frequency channels contained in the stream.
- **n_spectra_per_heap** (*int*) – The number of time samples received per frequency channel.
- **timestamp_step** (*int*) – Each heap contains a timestamp. The timestamp between consecutive heaps changes depending on the FFT size and the number of time samples per channel. This parameter defines the difference in timestamp values between consecutive heaps. This parameter can be calculated from the array configuration parameters for power-of-two array sizes, but is configurable to allow for greater flexibility during testing.
- **sample_bits** (*int*) – The number of bits per sample. Only 8 bits is supported at the moment.
- **heaps_per_fengine_per_chunk** (*int*) – Each chunk out of the SPEAD2 receiver will contain multiple heaps from each antenna. This parameter specifies the number of heaps per antenna that each chunk will contain.

property chunk_heaps: *int*

Number of heaps per chunk.

property heap_bytes

Calculate number of bytes in a heap based on layout parameters.

heaps_per_fengine_per_chunk: *int*

n_ants: *int*

n_channels_per_substream: *int*

n_spectra_per_heap: *int*

sample_bits: *int*

timestamp_step: *int*

katgpucbf.xbgpu.recv.**make_sensors**(*sensor_timeout: float*) → SensorSet

Create the sensors needed to hold receiver statistics.

Parameters

sensor_timeout – Time (in seconds) without updates before sensors for received data go into error and sensors for missing data becoming nominal.

katgpucbf.xbgpu.recv.**make_stream**(*layout:* Layout, *data_ringbuffer: ChunkRingbuffer*, *free_ringbuffer: ChunkRingbuffer*, *src_affinity: int*, *max_active_chunks: int*) → ChunkRingStream

Create a SPEAD receiver stream.

Helper function with XB-engine-specific logic in it.

Parameters

- **layout** – Heap size and chunking parameters.
- **data_ringbuffer** – Output ringbuffer to which chunks will be sent.
- **free_ringbuffer** – Ringbuffer for holding chunks for recycling once they've been used.
- **src_affinity** – CPU core affinity for the worker thread.
- **max_active_chunks** – Maximum number of chunks under construction.

async katgpucbf.xbgpu.recv.**recv_chunks**(*stream: ChunkRingStream*, *layout:* Layout, *sensors: SensorSet*, *time_converter:* TimeConverter) → AsyncGenerator[*Chunk*, None]

Retrieve chunks from the ringbuffer, updating metrics as they are received.

The returned chunks are yielded from this asynchronous generator.

Parameters

- **stream** – Stream object handling reception of F-engine data.
- **layout** – Structure of the stream.
- **sensors** – Sensor set containing at least the sensors created by *make_sensors()*.
- **time_converter** – Converter to turn data timestamps into sensor timestamps.

katgpucbf.xbgpu.xsend module

Module for sending baseline correlation products onto the network.

class katgpucbf.xbgpu.xsend.**Heap**(*context: AbstractContext*, *n_channels_per_substream: int*, *n_baselines: int*, *channel_offset: int*)

Bases: object

Hold all the data for a heap.

The content of the heap can change, but the class is frozen.

property timestamp: int

class katgpucbf.xbgpu.xsend.XSend(*output_name: str*, *n_ants: int*, *n_channels: int*, *n_channels_per_substream: int*, *dump_interval_s: float*, *send_rate_factor: float*, *channel_offset: int*, *context: AbstractContext*, *stream_factory: Callable[[StreamConfig, Sequence[ndarray]], spead2.send.asyncio.AsyncStream]*, *n_send_heaps_in_flight: int = 5*, *packet_payload: int = 8192*, *tx_enabled: bool = False*)

Bases: object

Class for turning baseline correlation products into SPEAD heaps and transmitting them.

This class creates a queue of buffers that can be sent out onto the network. To get one of these buffers call *get_free_heap()* - it will return a buffer. Once the necessary data has been copied to the buffer and it is ready to be sent onto the network, pass it back to this object using *send_heap()*. This object will create a limited number of buffers and keep recycling them - avoiding any memory allocation at runtime.

This has been designed to run in an asyncio loop, and *get_free_heap()* function makes sure that the next buffer in the queue is not in flight before returning.

To allow this class to be used with multiple transports, the constructor takes a factory function to create the stream.

Parameters

- **n_ants** – The number of antennas that have been correlated.
- **n_channels** – The total number of channels across all X-Engines. Must be a multiple of *n_channels_per_substream*.
- **n_channels_per_substream** – The number of frequency channels contained per substream.
- **dump_interval_s** – A new heap is transmitted every *dump_interval_s* seconds. Set to zero to send as fast as possible.
- **send_rate_factor** – Configure the spead2 sender with a rate proportional to this factor. This value is intended to dictate a data transmission rate slightly higher/faster than the ADC rate.

  Note: A factor of zero (0) tells the sender to transmit as fast as possible.

- **channel_offset** – Fixed value to be included in the SPEAD heap indicating the lowest channel value transmitted by this heap. Must be a multiple of *n_channels_per_substream*.
- **context** – All buffers to be transmitted will be created from this context.
- **stream_factory** – Callback function that will create the spead2 stream. It is passed the stream configuration and the memory buffers.
- **n_send_heaps_in_flight** – Number of buffers that will be queued at any one time. I don't see any need for this to be configurable, the data rates are likely too low for it to be an issue. I have put it here more to be explicit than anything else. This argument is optional.
- **packet_payload** – Size in bytes for output packets (baseline correlation products payload only, headers and padding are then added to this).
- **tx_enabled** – Start with output transmission enabled.

async get_free_heap() → *Heap*

Return a heap from the internal fifo queue when one is available.

There are a limited number of heaps in existence and they are all stored with a future object. If the future is complete, the buffer is not being used for sending and it will return the heap immediately. If the future is still busy, this function will wait asynchronously for the future to be done.

This function is compatible with asyncio.

Returns
Free heap

Return type
heap

header_size: Final[int] = 64

send_heap(*heap:* Heap) → None

Take in a buffer and send it as a SPEAD heap.

This function is non-blocking. There is no guarantee that a heap has been sent by the time the function completes.

Parameters
heap – Heap to send

async send_stop_heap() → None

Send a Stop Heap over the spead2 transport.

katgpucbf.xbgpu.xsend.**make_stream**(*, *output_name: str*, *dest_ip: str*, *dest_port: int*, *interface_ip: str*, *ttl: int*, *use_ibv: bool*, *affinity: int*, *comp_vector: int*, *stream_config: StreamConfig*, *buffers: Sequence[ndarray]*) → spead2.send.asyncio.AsyncStream

Produce a UDP spead2 stream used for transmission.

Module contents

20.2 Submodules

20.2.1 katgpucbf.curand_helpers module

Helpers to initialise random state with curand.

class katgpucbf.curand_helpers.**RandomStateBuilder**(*context: <Mock name='mock.Context' id='139785445150288'>*)

Bases: object

Build array of initialised random states for curand.

make_states(*shape: tuple[int, ...]*, *seed: int*, *sequence_first: int*, *sequence_step: int = 1*, *offset: int = 0*) → DeviceArray

Create a multi-dimensional array of random states.

This method is not particularly efficient. It's intended to be used just during startup, after which the random states will be persisted in global memory and reused.

20.2.2 katgpucbf.mapped_array module

20.2.3 katgpucbf.meerkat module

Constants applicable to the MeerKAT / MeerKAT Extension telescope.

class katgpucbf.meerkat.**Band**(*long_name: str*, *adc_sample_rate: float*, *centre_frequency: float*)

Bases: object

Holds presets for a known band.

adc_sample_rate: float

centre_frequency: float

long_name: str

20.2.4 katgpucbf.monitor module

Monitor classes allowing for rudimentary performance monitoring.

Queues in the form of asyncio.Queue are used for synchronisation between coroutines in *katgpucbf.fgpu*, but we may like to know a bit more about what's happening to them as items are pushed and popped. These metrics help us to see what bottlenecks there are, because if the queues get full (or the "free" queues get empty) it will result in dropped packets.

class katgpucbf.monitor.**FileMonitor**(*filename: str*)

Bases: *Monitor*

Write events to a file.

The file contains JSON-formatted records, one per line. Each record contains `time` and `type` keys, with additional type-specific information corresponding to the arguments to the notification functions.

close() → None

Close the output file cleanly.

event_qsize(*name: str*, *qsize: int*, *maxsize: int*) → None

Report the size and capacity of a queue.

The queue *name* has current size *qsize* and capacity *maxsize*. All calls with the same name must report the same *maxsize*.

event_qsize_delta(*name: str*, *delta: int*) → None

Report addition/removal of items from a queue.

The queue *name* has *delta* new items in it (or removed if *delta* is negative). This is an alternative to using *event_qsize()* when there is no easy way to obtain the absolute size of the queue. There must have been a previous call to *event_qsize()* to specify the initial capacity.

event_state(*name: str*, *state: str*) → None

Report the current state of a task.

The state `other` is conventional when no more specific information is available.

class katgpucbf.monitor.**Monitor**

Bases: ABC

Base class for performance monitors.

Subclasses can create *Queue* objects which report their size when it changes via the mechanism defined in the derived *Monitor* class.

Each subclass will need to override the abstract methods to record the performance events.

close() → None

Close the Monitor.

In the base class this does nothing, but if derived classes implement something that needs to close cleanly (such as an output file), then this function can be overridden to do that. It is called when you `__exit__` from using the *Monitor* as a context manager.

abstract event_qsize(*name: str*, *qsize: int*, *maxsize: int*) → None

Report the size and capacity of a queue.

The queue *name* has current size *qsize* and capacity *maxsize*. All calls with the same name must report the same *maxsize*.

abstract event_qsize_delta(*name: str*, *delta: int*) → None

Report addition/removal of items from a queue.

The queue *name* has *delta* new items in it (or removed if *delta* is negative). This is an alternative to using *event_qsize()* when there is no easy way to obtain the absolute size of the queue. There must have been a previous call to *event_qsize()* to specify the initial capacity.

abstract event_state(*name: str*, *state: str*) → None

Report the current state of a task.

The state `other` is conventional when no more specific information is available.

make_queue(*name: str*, *maxsize: int = 0*) → Queue

Create a *Queue* that reports its size via this *Monitor*.

time() → float

Get a timestamp, relative to the creation time of the monitor.

with_state(*name: str*, *state: str*, *return_state: str = 'other'*) → Generator[None, None, None]

Set a state for the duration of a block.

class katgpucbf.monitor.**NullMonitor**

Bases: *Monitor*

A do-nothing monitor that presents the required interface.

event_qsize(*name: str*, *qsize: int*, *maxsize: int*) → None

Report the size and capacity of a queue.

The queue *name* has current size *qsize* and capacity *maxsize*. All calls with the same name must report the same *maxsize*.

event_qsize_delta(*name: str*, *delta: int*) → None

Report addition/removal of items from a queue.

The queue *name* has *delta* new items in it (or removed if *delta* is negative). This is an alternative to using *event_qsize()* when there is no easy way to obtain the absolute size of the queue. There must have been a previous call to *event_qsize()* to specify the initial capacity.

event_state(*name: str*, *state: str*) → None

Report the current state of a task.

The state `other` is conventional when no more specific information is available.

make_queue(*name: str*, *maxsize: int = 0*) → Queue

Create a *Queue* that reports its size via this *Monitor*.

class katgpucbf.monitor.**Queue**(*monitor: Monitor*, *name: str*, *maxsize: int = 0*)

Bases: `Queue`

Extend `asyncio.Queue` with performance monitoring.

The only functionality added by any of the overridden functions is to call *event_qsize()* upon put/get events, transmitting an event to the parent *Monitor* object, alerting it about the change.

async get() → Any

Remove and return an item from the queue.

If queue is empty, wait until an item is available.

get_nowait() → Any

Remove and return an item from the queue.

Return an item if one is immediately available, else raise QueueEmpty.

async put(*item: object*) → None

Put an item into the queue.

Put an item into the queue. If the queue is full, wait until a free slot is available before adding item.

put_nowait(*item: object*) → None

Put an item into the queue without blocking.

If no free slot is immediately available, raise QueueFull.

20.2.5 katgpucbf.queue_item module

Provide *QueueItem*.

class katgpucbf.queue_item.**QueueItem**(*timestamp: int = 0*)

Bases: `object`

Queue Item for use in synchronisation between command queues.

Derived classes will have allocated memory regions associated with them, appropriately sized for input or output data. Actions (whether kernel executions or copies to or from the device) for these memory regions are initiated, and then an event marker is added to the list in some variation of this manner:

```
my_item.add_marker(command_queue)
```

The item can then be passed through a queue to the next stage in the program, which waits for the operations to be complete using *enqueue_wait_for_events()* or *async_wait_for_events()*. This indicates that the operation is complete and the next thing can be done with whatever data is in that region of memory.

add_marker(*command_queue: AbstractCommandQueue*) → AbstractEvent

Indicate that previous work on *command_queue* uses this item.

Future calls to *enqueue_wait_for_events()* or *async_wait_for_events()* will wait for all work issued to *command_queue* up to this point.

The associated event is returned.

async async_wait_for_events() → None

Wait for all events in the list of events to be complete.

enqueue_wait_for_events(*command_queue: AbstractCommandQueue*) → None

Block execution of a command queue until all of this item's events are finished.

Future work enqueued to *command_queue* will be sequenced after any work associated with the stored events.

property events: tuple[AbstractEvent, ...]

Get a copy of the currently registered events.

reset(*timestamp: int = 0*) → None

Reset the item's timestamp.

Subclasses should override this to reset other state. It is called by the constructor so it can also be used for initialisation.

timestamp: int

Timestamp of the item.

20.2.6 katgpucbf.recv module

Shared utilities for receiving SPEAD data.

class katgpucbf.recv.**BaseLayout**

Bases: ABC

Abstract base class for chunk layouts to derive from.

property chunk_bytes: int

Number of bytes per chunk.

abstract property chunk_heaps: int

Number of heaps per chunk.

chunk_place(*user_data: ndarray*) → LowLevelCallable

Generate low-level code for placing heaps in chunks.

Parameters
user_data – Data to pass to the placement callback

abstract property heap_bytes: int

Number of payload bytes per heap.

class katgpucbf.recv.**Chunk**(*self: spead2._spead2.recv.Chunk*, ***kwargs*)

Bases: Chunk

Collection of heaps passed to the GPU at one time.

It extends the spead2 base class to store a timestamp (computed from the chunk ID when the chunk is received), and optionally store a vkgdr device array.

When used as a context manager, it will call *recycle()* on exit.

device: object

recycle() → None

Return the chunk to the owning stream/group.

sink: ReferenceType

timestamp: int

katgpucbf.recv.**EVICTION_MODE = <EvictionMode.LOSSY: 0>**

Eviction mode to use when some streams fall behind

katgpucbf.recv.**RX_SENSOR_TIMEOUT_CHUNKS = 10**

Number of chunks before rx sensor status changes

katgpucbf.recv.**RX_SENSOR_TIMEOUT_MIN = 1.0**

Minimum rx sensor status timeout in seconds

class katgpucbf.recv.**StatsCollector**(*counter_map: ~collections.abc.Mapping[str, tuple[str, str]], labelnames: ~collections.abc.Iterable[str] = (), namespace: str = '', registry: ~prometheus_client.registry.CollectorRegistry = <prometheus_client.registry.CollectorRegistry object>*)

Bases: Collector

Collect statistics from spead2 streams as Prometheus metrics.

add_stream(*stream: ChunkRingStream | ChunkStreamGroupMember*, *labels: Iterable[str] = ()*) → None

Register a new stream.

If the collector was constructed with a non-empty `labelnames`, then `labels` must contain the same number of elements to provide the labels for the metrics that this stream will update.

> **Warning:** Calling this more than once with the same stream will cause that stream's statistics to be counted multiple times.

collect() → Iterable[Metric]

Implement Prometheus' Collector interface.

update() → None

Update the internal totals from the streams.

This is done automatically by *collect()*, but it can also be called explicitly. This may be useful to do just before a stream goes out of scope, to ensure that counter updates since the last scrape are not lost when the stream is garbage collected.

katgpucbf.recv.**add_reader**(*stream: ChunkRingStream | ChunkStreamGroupMember*, ***, *src: str | list[tuple[str, int]]*, *interface: str | None*, *ibv: bool*, *comp_vector: int*, *buffer: int*) → None

Connect a stream to an underlying transport.

See the documentation for Engine for an explanation of the parameters.

katgpucbf.recv.**make_stream**(***, *layout: BaseLayout*, *spead_items: list[int]*, *max_active_chunks: int*, *data_ringbuffer: ChunkRingbuffer*, *free_ringbuffer: ChunkRingbuffer*, *affinity: int*, *stream_stats: list[str]*, *user_data: ndarray*, *max_heap_extra: int = 0*, ***kwargs: Any*) → ChunkRingStream

Create a SPEAD receiver stream.

Parameters

- **layout** – Heap size and chunking parameters.

- **spead_items** – List of SPEAD item IDs to be expected in the heap headers.
- **max_active_chunks** – Maximum number of chunks under construction.
- **data_ringbuffer** – Output ringbuffer to which chunks will be sent.
- **free_ringbuffer** – Ringbuffer for holding chunks for recycling once they've been used.
- **affinity** – CPU core affinity for the worker thread (negative to not set an affinity).
- **stream_stats** – Stats to hook up to prometheus.
- **user_data** – Data to pass to the chunk placement callback
- **max_heap_extra** – Maximum non-payload data written by the place callback
- **kwargs** – Other keyword arguments are passed to `spead2.recv.StreamConfig`.

katgpucbf.recv.**make_stream_group**(*, *layout:* BaseLayout, *spead_items: list[int]*, *max_active_chunks: int*, *data_ringbuffer: ChunkRingbuffer*, *free_ringbuffer: ChunkRingbuffer*, *affinity: Sequence[int]*, *stream_stats: list[str]*, *user_data: ndarray*, *max_heap_extra: int = 0*, ***kwargs: Any*) → ChunkStreamRingGroup

Create a group of SPEAD receiver streams.

Parameters

- **layout** – Heap size and chunking parameters.
- **spead_items** – List of SPEAD item IDs to be expected in the heap headers.
- **max_active_chunks** – Maximum number of chunks under construction.
- **data_ringbuffer** – Output ringbuffer to which chunks will be sent.
- **free_ringbuffer** – Ringbuffer for holding chunks for recycling once they've been used.
- **affinity** – CPU core affinities for the worker threads (negative to not set an affinity). The length of this list determines the number of streams to create.
- **stream_stats** – Stats to hook up to prometheus.
- **user_data** – User data to pass to the chunk callback. It must have a field called *stats_base*, which will be filled in appropriately (modifying the argument).
- **max_heap_extra** – Maximum non-payload data written by the place callback
- **kwargs** – Other keyword arguments are passed to `spead2.recv.StreamConfig`.

20.2.7 katgpucbf.ringbuffer module

Wraps `spead2.recv.ChunkRingbuffer` with monitoring capabilities.

class katgpucbf.ringbuffer.**ChunkRingbuffer**(*self: spead2._spead2.recv.ChunkRingbuffer*, *maxsize: int*)

Bases: `ChunkRingbuffer`

Wraps `spead2.recv.ChunkRingbuffer` with monitoring capabilities.

When waiting for the next heap, it uses *`Monitor.with_state()`* to indicate that heaps are being waited for. Whenever a heap is retrieved, it updates the size of the queue.

async get() → Chunk

Override base class method to use the monitor.

20.2.8 katgpucbf.send module

Shared utilities for sending data over SPEAD.

class katgpucbf.send.**DescriptorSender**(*stream: spead2.send.asyncio.AsyncStream*, *descriptors: Heap*, *interval: float*, *first_interval: float | None = None*, ***, *substreams: Iterable[int] = (0,)*)

Bases: object

Manage sending descriptors at regular intervals.

The descriptors are first sent once immediately, then after *first_interval* seconds, then every *interval* seconds. Using a different *first_interval* makes it possible to stagger different senders so that their descriptors do not all arrive at a common receiver at the same time.

The descriptors are sent with zero rate, which means they will not affect the timing of other packets in the same stream.

Parameters

- **stream** – The stream to which the descriptor will be sent. It will be sent to all substreams simultaneously.
- **descriptors** – The descriptor heap to send.
- **interval** – Interval (in seconds) between sending descriptors.
- **first_interval** – Delay (in seconds) immediately after starting. If not specified, it defaults to *interval*.
- **substreams** – Substream indices to which descriptors are sent. If not specified, send only to the first substream.

halt() → None

Request *run()* to stop, but do not wait for it.

async run() → None

Send the descriptors indefinitely (use *halt()* or cancel to stop).

20.2.9 katgpucbf.spead module

Common SPEAD-related constants and helper function.

katgpucbf.spead.**DEFAULT_PORT = 7148**

Default UDP port

katgpucbf.spead.**DIGITISER_STATUS_SATURATION_COUNT_SHIFT = 32**

First bit position in digitiser status SPEAD item for ADC saturation count

katgpucbf.spead.**DIGITISER_STATUS_SATURATION_FLAG_BIT = 1**

Bit position in digitiser_status SPEAD item for ADC saturation flag

katgpucbf.spead.**FLAVOUR = <spead2._spead2.Flavour object>**

SPEAD flavour used for all send streams

katgpucbf.spead.**IMMEDIATE_DTYPE = dtype('>u8')**

dtype for items that need to be immediate yet passed by reference

katgpucbf.spead.**IMMEDIATE_FORMAT = [('u', 48)]**

Format for immediate items

katgpucbf.spead.**make_immediate**(*id: int*, *value: Any*) → Item

Synthesize an immediate item.

Parameters

- **id** – The SPEAD identifier for the item
- **value** – The value of the item

20.2.10 katgpucbf.utils module

A collection of utility functions for katgpucbf.

class katgpucbf.utils.**DeviceStatusSensor**(*target: SensorSet*, *name: str = 'device-status'*, *description: str = 'Overall engine health'*)

Bases: SimpleAggregateSensor[DeviceStatus]

Summary sensor for quickly ascertaining device status.

This takes its value from the worst status of its target set of sensors, so it's quick to identify if there's something wrong, or if everything is good.

aggregate_add(*sensor: Sensor[_T]*, *reading: Reading[_T]*) → bool

Update internal state with an additional reading.

Returns
True if the new reading should result in a state update.

Return type
bool

aggregate_compute() → tuple[Status, DeviceStatus]

Compute aggregate status and value from the internal state.

aggregate_remove(*sensor: Sensor[_T]*, *reading: Reading[_T]*) → bool

Update internal state by removing a reading.

Returns
True if removing the reading should result in a state update.

Return type
bool

filter_aggregate(*sensor: Sensor*) → bool

Decide whether another sensor is part of the aggregation.

Users can override this function to exclude certain categories of sensors, such as other aggregates, to prevent circular references.

Returns
True if *sensor* should be included in calculation of the aggregate, False if not.

Return type
bool

update_aggregate(*updated_sensor: Sensor[_T] | None*, *reading: Reading[_T] | None*, *old_reading: Reading[_T] | None*) → Reading[DeviceStatus] | None

Update the aggregated sensor.

The user is required to override this function, which must return the updated `Reading` (i.e. value, status and timestamp) which will be reflected in the *Reading* of the aggregated sensor.

Parameters

- **updated_sensor** – The sensor in the target `SensorSet` which has changed in some way.
- **reading** – The current reading of the *updated_sensor*. This is *None* if the sensor is being removed from the set.
- **old_reading** – The previous reading of the *updated_sensor*. This is *None* if the sensor is being added to the set.

Returns
The reading (value, status, timestamp) that should be shown by the *AggregatedSensor* as a result of the change. If None is returned, the sensor's reading is not modified.

Return type
Optional[Reading]

class katgpucbf.utils.**TimeConverter**(*sync_time: float*, *adc_sample_rate: float*)

Bases: `object`

Convert times between UNIX timestamps and ADC sample counts.

Note that because UNIX timestamps are handled as 64-bit floats, they are only accurate to roughly microsecond precision, and will not round-trip precisely.

Parameters

- **sync_time** – UNIX timestamp corresponding to ADC timestamp 0
- **adc_sample_rate** – Number of ADC samples per second
- **todo::** (`..`) – This does not yet handle leap-seconds correctly.

adc_to_unix(*samples: float*) → float

Convert an ADC sample count to a UNIX timstamp.

unix_to_adc(*timestamp: float*) → float

Convert a UNIX timestamp to an ADC sample count.

class katgpucbf.utils.**TimeoutSensorStatusObserver**(*sensor: Sensor*, *timeout: float*, *new_status: Status*)

Bases: `object`

Change the status of a sensor if it doesn't receive an update for a given time.

Do not directly attach or detach this observer from the sensor (it does this internally). It is not necessary to retain a reference to the object unless you wish to interact with it later (for example, by calling `cancel()`).

It must be constructed while there is a running event loop.

cancel() → None

Detach from the sensor and make no further updates to it.

katgpucbf.utils.**add_gc_stats**() → None

Add Prometheus metrics for garbage collection timing.

It is only safe to call this once.

katgpucbf.utils.**add_signal_handlers**(*server: DeviceServer*) → None

Arrange for clean shutdown on SIGINT (Ctrl-C) or SIGTERM.

katgpucbf.utils.**add_time_sync_sensors**(*sensors: SensorSet*) → Task

Add a number of sensors to a device server to track time synchronisation.

This must be called with an event loop running. It returns a task that keeps the sensors periodically updated.

katgpucbf.utils.**comma_split**(*base_type: Callable[[str], _T]*, *count: int | None = None*, *allow_single=False*) → Callable[[str], list[_T]]

Return a function to split a comma-delimited str into a list of type _T.

This function is used to parse lists of CPU core numbers, which come from the command-line as comma-separated strings, but are obviously more useful as a list of ints. It's generic enough that it could process lists of other types as well though if necessary.

Parameters

- **base_type** – The base type of thing you expect in the list, e.g. *int*, *float*.
- **count** – How many of them you expect to be in the list. *None* means the list could be any length.
- **allow_single** – If true (defaults to false), allow a single value to be used when *count* is greater than 1. In this case, it will be repeated *count* times.

katgpucbf.utils.**gaussian_dtype**(*bits: int*) → dtype

Get numpy dtype for a Gaussian (complex) integer.

Parameters

bits – Number of bits in each real component

katgpucbf.utils.**parse_source**(*value: str*) → list[tuple[str, int]] | str

Parse a string into a list of IP endpoints.

katgpucbf.utils.**steady_state_timestamp_sensor**() → Sensor[int]

Create `steady-state-timestamp` sensor.

20.3 Module contents

katgpucbf.**DEFAULT_PACKET_PAYLOAD_BYTES: Final = 8192**

Biggest power of 2 that fits in a jumbo MTU. A power of 2 isn't required but it can be convenient to have packet boundaries align with the natural boundaries in the payload (for antenna-channelised-voltage output). Bigger is better to minimise the number of packets/second to process.

katgpucbf.**DEFAULT_TTL: Final = 4**

Default TTL for spead multicast transmission

katgpucbf.**MIN_SENSOR_UPDATE_PERIOD: Final = 1.0**

Minimum update period (in seconds) for katcp sensors where the underlying value may update extremely rapidly.

BIBLIOGRAPHY

[Merry2023] Bruce Merry, "Efficient channelization on a graphics processing unit," J. Astron. Telesc. Instrum. Syst. 9(3) 038001 (12 July 2023). https://doi.org/10.1117/1.JATIS.9.3.038001

[Pri] Danny C. Price. Spectrometers and polyphase filterbanks in radio astronomy. arXiv:1607.03579.

PYTHON MODULE INDEX

k

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z