gCorr – Data structures and processing flow

v0.2 16/4/18

Processing approach

Data processing for gCorr happens in 5 main steps

1. Telescope data is read from disk. Data is assumed to be binary and contain no headers. Data format is the same as described in the "Packed Data" below.
2. Raw data (encoded as integers) is converted to floating point. For convenience conversion creates a complex number with zero imaginary value (phase 0). Data is split into independent channels at this stage. "Delay" correction is also applied at this stage to correct for geometric effects of the telescopes receiving signals at different times (and errors in local clocks)
3. *Each sample is "fringe" rotated to account for different velocities of the telescopes compared to each other (doppler shift). This is achieved by applied a time varying phase offset to each sample (not implemented yet).*
4. "N" time samples are Fourier transformed to "channelized" the data. This is repeated for each "N" samples in time
5. For each FFT block the individual frequency channels from each telescope and multiplied and then accumulated to form "visibilities". For N antennas there are N(N-1)/2 unique baseline combinations. Each combination must be formed. A data from series of FFT are accumulated to form a "sub integration". Typically this will be 10s to 100s millisec. Sub integrations will finally be averaged to about a second to form the final visibility integration.

gCorr will read and process data in blocks of "`numffts`" FFTs, each FFT is of length "`fftchannel`". This will produce visibilities with "`numchannel`" unique frequency points. For real sampled data, "`fftchannels`" is twice "`numchannels`". For complex sampled data "`fftchannels`" equals "`numchannels`". For examples if "`numchannels`" equals 1024 and "`numffts`" equals 3250, data is processed in blocks of 6656000 samples (~6.6 million time samples).

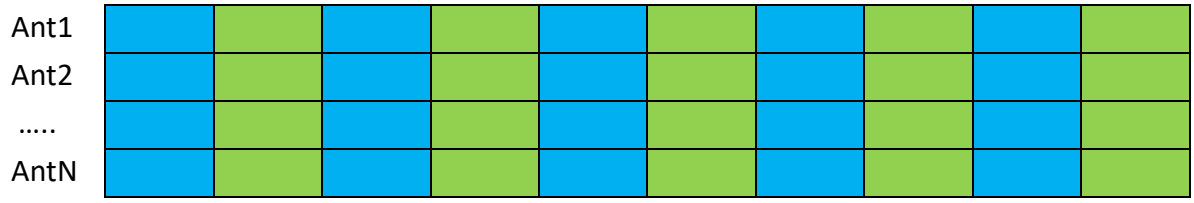
Note gCorr uses 2D C style arrays (pointer to an array of pointers). When memory is allocated on the GPU, cudaMalloc returns a pointer to the CPU to the allocated memory. This makes it impossible to transparently access the 2D array of data from cuda kernels and CPU based functions (cuFFT, cudaMemset). To work around this, 2 sets of pointers are needed. One for the GPU and one for the CPU.

Packed Data

Initial data is raw voltages sampled as integers. Voltages can be encoded as integers with bit depths of 1-16 bits. If the number of bits is >4 either unsigned integers or 2s complement s used. For 2 bit data a variety of different encodings of the 4 states can be used. Voltages samples can be either real or complex numbers. Usually data from a telescope has been subdivided into multiple channels. For examples 64 MHz of bandwidth may be subdivided into four 16 MHz channels. Further to this, typically 2 orthogonal polarisations are recorded. From a data structure perspective polarisations are treated as further channels.

When the #bits/samples is < 8, samples are usually packed tightly within a byte. For example for a 2 bit, 2 channel setup each byte will contain 4 samples. The least significant 4 bits contain 2 time samples of the 2 channels, the next 4 bits contain the next time sample of each channel.

For simplicity, gCorr only supports a 2 polarisation data (2 channels, each of the same frequency but orthogonal polarisations). 2 bit data is assumed, though this may be expanded to 8 bit data. Raw data is read, telescope by telescope into a "C" 2-D array (pointer to array of pointers) of unsigned 8bit integers (bytes) (`**inputdata`)
E.g.



For each telescope, and with 2bit real voltages with 2 polarisations, the data is assumed to be packed as (each colour represents a single byte). "A" and "B" represent the 2 polarisations and A0 is the first time sampled, A1 the second time sample etc.

For 8 bit data, with 2 polarisations, each data from each telescopes is encoded as:

Unpacked Data

The first step is to convert data from packed integers into floating point numbers. Each telescope is unpacked separately – the full sub-integration of data is unpacked in a single kernel call (per telescope). Unpacked data is stored as a 2D complex float array (`**unpackedData, **unpackedData_h`). The first axis is antenna index and polarisation(`numantennas*2` values), the second is polarisations (2 values) and the third voltage time samples (`fftchannels` values). The imaginary part of the complex number is set to zero.

The unpacker also makes an adjustment for timing differences between telescopes. The unpacked data array looks like. Not each Pol for each telescope is a separate row in the array. Each colour represents a complex float (ie 8 bytes).

Ant1 Pol A	A0 Real	A0 Imag	A1 Real	A1 Imag	…	AN Real	AN Imag
Ant1 Pol B	B0 Real	B0 Imag	B1 Real	B1 Imag	…	BN Real	BN Imag
Ant2 Pol A	A0 Real	A0 Imag	A1 Real	A1 Imag	…	AN Real	AN Imag
Ant2 Pol B	B0 Real	B0 Imag	B1 Real	B1 Imag	…	BN Real	BN Imag

….	…	…	…	…	…	…	…
AntN Pol A	A0 Real	A0 Imag	A1 Real	A1 Imag	…	AN Real	AN Imag
AntN Pol B	B0 Real	B0 Imag	B1 Real	B1 Imag	…	BN Real	BN Imag

(N = fftchannels)

The unpacker kernel unpacks a single byte per thread.

Fringe rotation

To compensate for velocity difference between stations, each time sample has to have a phase correction. The phase variation is assumed to be linear across the length of an FFT. This has not been added, though untested code in `gxkernel.cu` has been written.

FFT

Data is then FFT'ed for each station/polarisation. For (originally) real sampled data half the output channels of the FFT contain no information – this is discarded. For (originally) complex sampled data all output channels are used. At this stage we have "`numchannel`" frequency points per telescope/polarisation. The output data from the FFT is packed into a 2D array of complex floats (`**channelisedData`). The first axis is antenna index (`numantennas` values), the second is polarisations (2 values) and the third frequency points (`fftchannels` values).

Ant 1 Pol A	A0	A1	A2	A3	…	AN
Ant 1 Pol B	B0	B1	B2	B3	…	BN
Ant 2 Pol A	A0	A1	A2	A3	…	AN
Ant 2 Pol B	B0	B1	B2	B3	…	BN
….	…	…	…	…	…	…
Ant M Pol A	A0	A1	A2	A3	…	AN
Ant M Pol B	B0	B1	B2	B3	…	BN

(N = fftchannels). Each colour represents a single complex float value (8 bytes).
Data is processed per telescope/polarisation in a look calling `cufftExecC2C`. All data for this station/polarisation is FFTed in a batch per loop.

Fractional Delay Correction

Previous time delay compensation (by shifting data stream in samples) has an error up to +-0.5 samples. This corresponds to a phase gradient of 180 degrees across the band. This needs to be compensated after the FFT with a linear phase correction. This is not implemented currently.

Cross Correlation

For each telescope combination the matching FFT output channel must be multiplied to achieve the interferometric correlation. For `numantennas` antennas, there are `numantennas*(numantennas-1)/2` combinations (baselines) – `nbaselines`. Because the we are also interested in the cross polarisation correlation, the independent

polarisations from each telescope is also multiped. Cross correlation products are stored in a 2D complex float array.). Each kernel thread does a single frequency channel, and all baseline/polarisation combinations for that frequency channel. For efficiency, baseline accumulation is done at the same time. To avoid memory conflicts on the accumulation, and to keep the thread count high, accumulation is done in multiple parallel batches (`parallelAccum`) and each thread correlates and accumulates multiple time samples (`nchunk`)

THE FOLLOWING MAY BE WRONG AND NEEDS TO BE DOUBLE CHECKED. The corresponding code also!
Correlated and accumulated data is stored in 2D array (`**channelisedData`, `**channelisedData_h`). The first axis is antenna index, polarisation product AND parallel accumulations (`numantennas*4*parallelAccum` values), the second is polarisations points (`numchannel` values).

Ant 1-2 Pol AA	AA0	AA1	AA2	AA3	…	AAN
Ant 1-2 Pol AB	AB0	AB1	AB2	AB3	…	ABN
Ant 1-2 Pol BA	BA0	BA1	BA2	BA3	…	BAN
Ant 1-2 Pol BB	BB0	BB1	BB2	BB3	…	BBN
Ant 1-3 Pol AA	AA0	AA1	AA2	AA3	…	AAN
Ant 1-3 Pol AB	AB0	AB1	AB2	AB3	…	ABN
Ant 1-3 Pol BA	BA0	BA1	BA2	BA3	…	BAN
Ant 1-3 Pol BB	BB0	BB1	BB2	BB3	…	BBN
….	…	…	…	…	…	…
Ant N-M Pol AA	AA0	AA1	AA2	AA3	…	AAN
Ant N-M Pol AB	AB0	AB1	AB2	AB3	…	ABN
Ant N-M Pol BA	BA0	BA1	BA2	BA3	…	BAN
Ant N-M Pol BB	BB0	BB1	BB2	BB3	…	BBN
Repeat of block above for next time slice						
etc						

Final Accumulation

The `parallelAccum` blocks finally need to be average together. The kernel to do this runs each frequency point as a separate thread but attempts no further parallelisation on the assumption that there is minimal compute to be saved.