2017-12-01

# Real-Time Beamforming Algorithms for the Focal L-Band Array on the Green Bank Telescope

Mark William Ruzindana
*Brigham Young University*

# Real-Time Beamforming Algorithms for the Focal L-band Array on the Green Bank Telescope (FLAG)

Mark William Ruzindana

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Karl F. Warnick, Chair
Brian D. Jeffs
David G. Long

Department of Electrical and Computer Engineering

Brigham Young University

ABSTRACT

Real-Time Beamforming Algorithms for the Focal L-band Array
on the Green Bank Telescope (FLAG)

Mark William Ruzindana
Department of Electrical and Computer Engineering, BYU
Master of Science

A phased array feed (PAF) provides a contiguous, electronically synthesized wide field of view for large-dish astronomical observatories. Significant progress has been made in recent years in improving the sensitivity of PAF receivers though optimizing the design of the antenna array, cryogenic cooling of the front end, and implementation of real-time correlation and beamforming in digital signal processing.

FLAG is a 19 dual-polarized element phased array with cryogenic LNAs, direct digitization of RF signals at the front end, digital signal transport over fiber, and a real time signal processing back end with up to 150 MHz bandwidth. The digital back end includes multiple processing modes, including real-time beamforming, real-time correlation, and a separate real-time beamformer for commensal radio transient searches. Following a polyphase filterbank operation performed in field programmable gate arrays (FPGAs), beamforming, correlation, and integration are implemented on graphical processing units (GPUs) that perform parallelized operations. Parallelization greatly increases processing speed and allows for real-time signal processing.

During a recent test/commissioning of FLAG, Tsys/efficiency of approximately 28 K was measured across the PAF field of view and operating bandwidth, corresponding to a system temperature below 20 K. To demonstrate the astronomical capability of the receiver, a pulsar (PSR B1937+21) was detected with the real-time beamformer.

This thesis provides details on the development of the FLAG digital back end, the real-time beamformer, and reports on the commissioning tests of the FLAG PAF receiver developed by the National Radio Astronomy Observatory (NRAO), Green Bank Observatory (GBO), West Virginia University (WVU), and Brigham Young University for the Green Bank Telescope (GBT).

Keywords: phased array feed, correlation, beamforming, digital signal processing

# ACKNOWLEDGMENTS

I would like to thank my family for supporting me through University. I would not have been able to do anything that I am doing know without their support. My parents, Augustine, and Colette Ruzindana, are the best people that I know, and inspired me to be a better person everyday of my life. My sisters; Rose, Belinda, and Aster, have been very supportive, and I can't thank them enough.

My colleagues, Richard Black, Mitchell Burnett, and Junming Diao, were very helpful in the radio astronomy research group, and are now some of my best friends. It was so nice to have people to work with through my research, and I would like to think that we helped each other become better engineers, and people.

Lastly, I would like to thank my committee, Karl Warnick, Brian Jeffs, and David Long, as well as all the professors that have helped me through my Master's degree. My advisor, Dr. Karl Warnick has been very supportive and has helped me whenever I needed it. My work in radio astronomy would not have been possible without him. Dr. Brian Jeffs has also been very helpful with my research in radio astronomy, and was in a way a co-advisor. And Dr. David Long, as well as other professors, have been so helpful in my pursuit of higher education.

I am extremely grateful for all the support and help that has been provided over the past few years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

# CHAPTER 1. INTRODUCTION

## 1.1 Motivation

A phased array feed (PAF) is a 2-D planar array of antennas (order of 1/2 wavelength apart) placed at the focus of a large telescope dish that spatially samples the focal plane. PAFs are ideal for wide-field and survey instruments where it is desired to cover large fields of view over short periods of time [1], [2].

PAFs are primarily used to steer multiple simultaneous beams across a field of view and this is referred to as beamforming [3]–[6]. In radio astronomy (the study of radio sources in deep space), this field of view is the sky, and is much larger than would be achieved with the traditional single pixel horn feed [1], [2]. Due to the large field of view, the survey speed also increases. Other advantages of PAFs include sensitivity optimization and radio frequency interference mitigation.

PAFs are being developed to detect radio sources such as transients which move rapidly over time. This requires real-time beamforming to detect transients such as pulsars or fast radio bursts (FRBs) [7]–[9]. Real-time is achieved when the computation speed is less than or equal to the sample rate. The beamforming algorithms can be executed on devices such as a field programmable gate array (FPGA) or a graphical processing unit (GPU). While FPGAs have proven to be capable of such speeds, GPUs are faster and computationally more efficient [10].

The Focal L-band Array for the Green Bank Telescope (FLAG) is a wide-field astronomical PAF receiver with broadband signal processing and operational science observation capability [11]. The backend of this instrument is currently in development by Brigham Young University (BYU) and West Virginia University (WVU). This instrument is expected to perform fine/coarse channel correlation as well as real-time beamforming. This thesis describes the real-time beamforming algorithms implemented on GPUs, as well as the overall development of the backend system, and the results obtained from the Green Bank Observatory (GBO).

## 1.2 Related Work

At Brigham Young University, the x64 DAQ digital receiver system, a 20 MHz digital signal processing (DSP) module for data acquisition, was developed for a 64 element PAF [12]. The system FPGA firmware was developed using the Collaboration for Astronomy Signal Processing and Electronics Research (CASPER) Reconfigurable Open-Architecture Hardware (ROACH) environment [13], [14].

All 64 elements were frequency channelized to DFT lengths of 256, 512, and 1024. With a 512-length DFT, up to 59 frequency channels across all 64 ADC inputs were streamed to disk on a server PC. This system was deployed at the Arecibo observatory and was tested with a 19-element dual-polarized PAF created by Cornell University. The team was able to measure critical performance parameters of the Cornell feed, such as sensitivity, system noise temperature, and beamwidths.

Prior to the x64 system, a group of graduate students from BYU, Vikas Asthana, Taylor Webb, and Mike Elmer, developed two PAF digital back ends both supporting an analog bandwidth of 450 KHz [1], [15], [16]. Initially, a 20-input back end was developed for a 19-element single-polarization thickened dipole PAF by Asthana and Elmer. This back end was then modified to 40 inputs when the group completed their first 19-element dual-polarization array. This backend was developed by Webb and Elmer.

Both systems were deployed at the Arecibo Observatory, and while accurate and stable results were obtained, data could be acquired for approximately 60 seconds at most. The 450 KHz bandwidth was also too small to be useful to astronomers.

The x64 and 450 KHz systems were the predecessors of the digital backend designed for FLAG. The development of these systems enabled the design of a 150 MHz back end capable of acquiring data over long periods of time.

## 1.3 Contributions

The contributions made by the author are summarized in the following list:

- Implemented signal processing code that enabled pulsar detection with the first cryogenic astronomical phased array feed (PAF).

- Conducted experiments in Green Bank, West Virginia, that enabled the measurement of a $T_{\text{sys}}/\eta$ of 28 K, comparable to that of a single pixel horn feed.

## 1.4 Thesis Outline

Chapter 2, discusses the theory behind the processing algorithms used for the digital back end for FLAG (PAF beamforming, calibration, and sensitivity calculation).

Chapter 3 provides a FLAG system overview and discusses the work that went into the development of the real-time beamformer for FLAG.

Chapter 4, 5, and 6 discuss the tests that were run on the system, the overall development of the back end, and the results obtained from the Green Bank observatory. Each of these tests was called a commissioning, and involved several hours of data acquisition split over two to three weeks.

Chapter 7, offers a summary on the results obtained and work that could be done in the future.

# CHAPTER 2. BACKGROUND

This chapter discusses the mathematics, algorithms, and basic theory necessay to understand the remainder of this thesis. It is not meant to be exhaustive, but to provide a basic understanding of the theory behind PAF beamforming, calibration, and sensitivity calculation used in the digital back end for FLAG.

An introduction to radio astronomy practices and theory can be found [17], [18]. For more details on beamforming, and array signal processing, refer to [3]–[6].

## 2.1 PAF Beamforming

As mentioned in the previous chapter, beamforming is a signal processing technique used by PAFs to form multiple simultaneous beams over a field of view [3]–[6]. Figure 2.1 shows the structure of a beamformer.

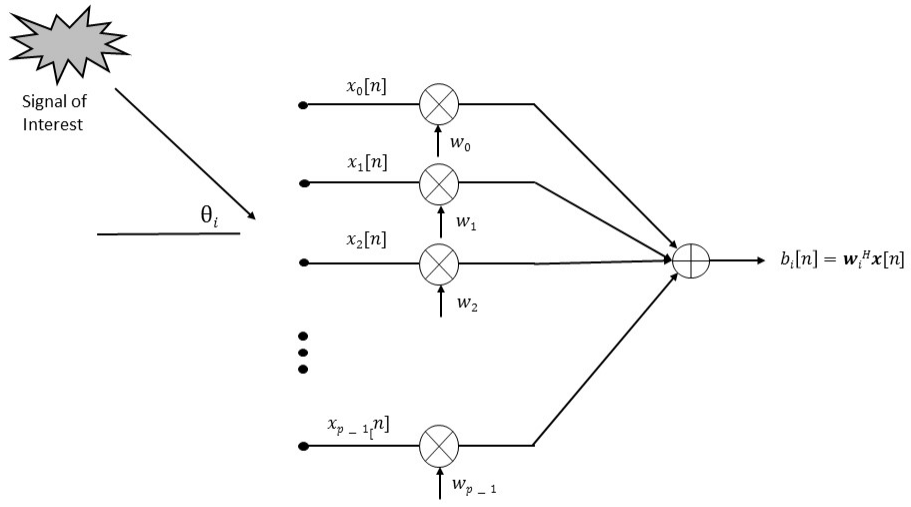



Figure 2.1: The structure of a beamformer with a $P$ element PAF generating multiple simulataneous beams.

The $P$ element PAF produces a length $P \times 1$ data vector, $\mathbf{x}[n]$, at time sample $n$. The array sample voltage signal vector, $\mathbf{x}[n]$, is complex baseband bandshifted, and is modeled as

$$\mathbf{x}[n] = \mathbf{a}(\theta_i)s[n] + \boldsymbol{\eta}[n], \tag{2.1}$$

where $s[n]$ is a random process that models the baseband volatge of the signal of interest (SOI) at a given time sample, n, $\mathbf{a}(\theta_i)$ is a normalized array response to a unit amplitude point source in the far field at directions, $\theta_i$, corresponding to $s[n]$, and $\boldsymbol{\eta}[n]$ is the complex random vector representing the noise.

Using weight vectors computed with the calibration procedure detailed in the next section, the beamformed output is given by

$$b_i[n] = \mathbf{w}_i^H \mathbf{x}[n], \tag{2.2}$$

where $\mathbf{w}_i$ is the vector of complex weights required to steer the beam to the $i$th direction, and $^H$ is the conjugate transpose of the vector.

Beamformer weights can be calculated in various ways depending on the application. Some of the algorithms used to calculate weights include; maximum signal to noise ratio (SNR), minimum variance distortionless response (MVDR), linearly constrained minimum variance (LCMV), subspace projection and others. When calculating the weights used in this thesis, the maximum SNR algorithm was chosen in order to produce beamformed output with high SNR. The signal and noise covariance matrices were also available, and were used for calibration as well as SNR calculation [3]–[6].

## 2.2 PAF Calibration

In order to calculate beamformer weights, a calibration scan is performed on a known source. The calibration is performed by steering the dish across the source and calculating covariance matrices at each point [9]. An "on" covariance matrix is calculated from "on" pointings (dish steered across the source), and an "off" covariance matrix from the "off pointings" (data steered off of the source). These covariance matrices are calculated as follows

$$\mathbf{R}_x = E\{\mathbf{x}[n]\mathbf{x}^H[n]\}, \tag{2.3}$$

$$\mathbf{R}_\eta = E\{\boldsymbol{\eta}[n]\boldsymbol{\eta}^H[n]\}, \tag{2.4}$$

where $\mathbf{R}_x$ is the "on" covariance matrix (the sample spatial covariance matrix for both signal and noise), $\mathbf{R}_\eta$ is the "off" covariance matrix (noise covariance matrix), and $E\{\}$ is the expectation. Figure 2.2 shows an example of a 6×6 calibration grid used to calculate the covariance matrices.

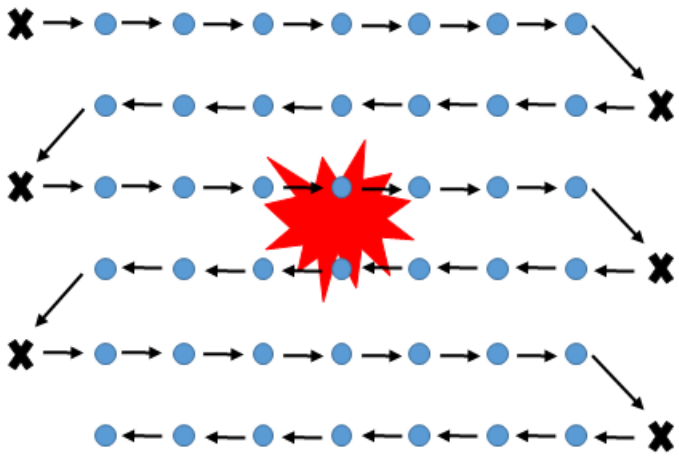


Figure 2.2: An example of a 6×6 calibration grid with 36 on-pointings, 6 off-pointings, and the source in the center of the grid.

The steering vectors corresponding to each point on the grid were estimated using the following

$$\mathbf{R}_\eta^{-1}\hat{\mathbf{R}}_x\mathbf{R}_\eta^{-1}\hat{\mathbf{a}}(\theta_i) = \lambda\mathbf{R}_\eta^{-1}\hat{\mathbf{a}}(\theta_i), \tag{2.5}$$

where $\hat{\mathbf{a}}(\theta_i)$ is the estimated steering vector. Let $\tilde{\mathbf{a}}(\theta_i) = \mathbf{R}_\eta^{-1}\hat{\mathbf{a}}(\theta_i)$, then

$$\mathbf{R}_\eta^{-1}\hat{\mathbf{R}}_x\tilde{\mathbf{a}}(\theta_i) = \lambda\tilde{\mathbf{a}}(\theta_i). \tag{2.6}$$

Using eigen decomposition, $\tilde{\mathbf{a}}(\theta_i)$ is found to be the dominant eigenvector. Therefore, the estimated steering vector is

$$\hat{\mathbf{a}}(\theta_i) = \mathbf{R}_\eta \tilde{\mathbf{a}}(\theta_i), \tag{2.7}$$

After estimating the steering vectors, the max-SNR beamformer weights can be calculated by

$$\mathbf{w}_i = \mathbf{R}_\eta^{-1} \hat{\mathbf{a}}(\theta_i), \tag{2.8}$$

## 2.3 Sensitivity Calculation

With the weights calculated, if we let $P_x = \mathbf{w}^H \mathbf{R}_x \mathbf{w}$, the signal plus noise power, and $P_\eta = \mathbf{w}^H \mathbf{R}_\eta \mathbf{w}$, the noise power, we can calculate the SNR using the following equation,

$$\text{SNR} = \frac{P_x - P_\eta}{P_\eta}. \tag{2.9}$$

The sensitivity can then be calculated by

$$\text{SNR} = \frac{10^{-26} \frac{1}{2} F_\text{s} A_\text{e} B}{k_\text{B} T_\text{sys} B}, \tag{2.10}$$

$$\frac{A_\text{e}}{T_\text{sys}} = \frac{2k_\text{B}}{10^{-26} F_\text{s}} \text{SNR}, \tag{2.11}$$

where $A_\text{e}$ is the effective aperture area, $T_\text{sys}$ is the system temperature and the ratio between them is the sensitivity. $k_\text{B}$ is Boltzmann's constant ($\sim 1.38 \times 10^{-23} \text{m}^2 \text{ kg s}^{-2} \text{ K}^{-1}$), $B$ is the bandwidth, and $F_\text{s}$ is the signal flux density of one polarization of the source in Jansky (Jy) [3]–[6].

## 2.4 Summary

Using the theory and calculations from this chapter, we can process and analyze the data acquired with FLAG. The data is processed with a digital back end comprising of digital signal processing boards and high performance computers among other devices. These devices will be discussed in the following chapter starting with a FLAG system overview.

# CHAPTER 3. FLAG SYSTEM AND BEAMFORMER IMPLEMENTATION

In radio astronomy, in order to detect transient sources with a PAF, beamformer computational speed must be fast enough to keep with the rotational period of the source. The FLAG system achieves this speed with the use of GPUs. Each GPU processed $1/10$ of the bandwidth. This is described in more detail in the following sections.

This chapter discusses the theory and work done to achieve real-time beamforming with FLAG. It first describes the FLAG system on a high-level. The front end is briefly described since this thesis focuses on testing as well as work done on the digital back end.

## 3.1 FLAG System Overview

FLAG is a 19 element, dual-polarization, cryogenic PAF with direct digitization of RF signals at the front end, digital signal transport over fiber, and a real time signal processing back end with up to 150 MHz bandwidth [11]. This system has new science observation capability, and will be used to discover more than 50 new pulsars within the inner galactic plane and study diffuse HI around galaxies [11]. A high-level block diagram of the system can be seen in Figure 3.1.

The PAF receiver front end has a 19 element dual-polarization array (as previously mentioned), cryogenically cooled low noise amplifiers (LNAs), and a digital down converter [19], [20]. The receiver can be seen in Figure 3.2 where it was placed in the outdoor test facility at GBO.

The FLAG back end is located in the Green Bank Telescope (GBT) equipment room called the Jansky lab. It consists of five digital optical receiver cards, five ROACH II FPGA boards [14], [21], an ethernet switch, and five HPCs (See appendix A). These parts are all connected in the order listed. A picture of the back end can be seen in Figure 3.3. Specifications of theses devices are provided in the appendix.

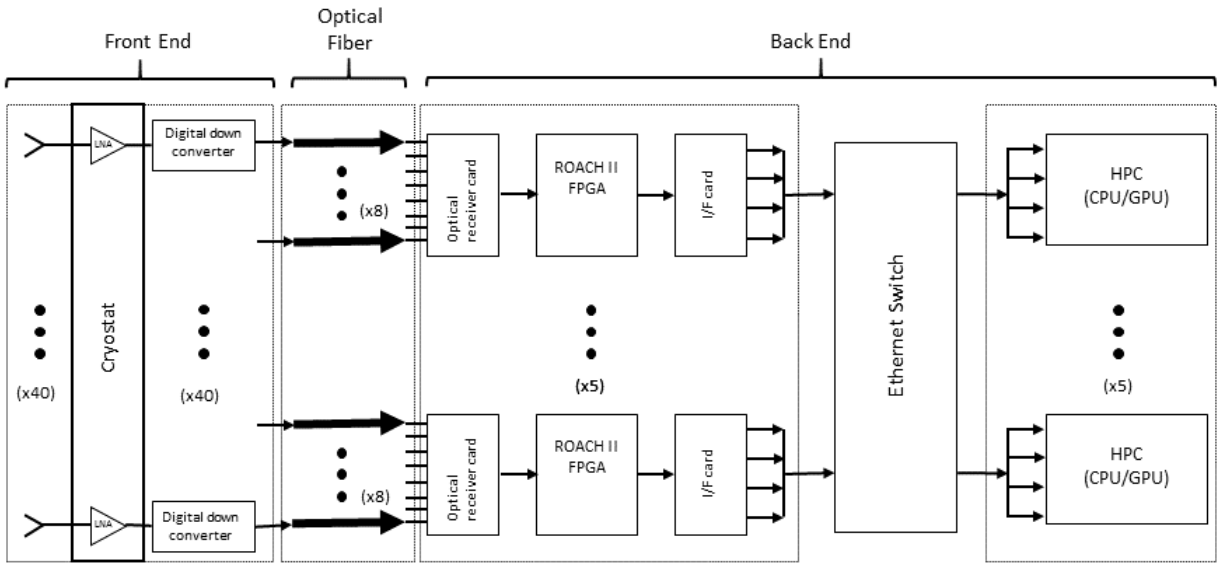



Figure 3.1: A high-level block diagram of FLAG showing details on the front end, and back end.

The digitized signals from the front end are serialized and sent over 40 optical fibers [22] to the optical receiver cards which are connected to the ROACH II boards. The boards channelize the approximately 150 MHz bandwidth into 512 channels each with a bandwidth of 303 KHz.

The data is then reduced to 500 frequency channels and packetized into 10 user-datagram protocol (UDP) packets each containing 50 frequency samples for eight antennas across 20 time samples. These packets are streamed over 10-Gbe/40-Gbe breakout cables into a 12-port 40-GbE network switch, which redirects packets into the HPCs such that each one receives 100 frequency samples for all 40 antennas.

Each HPC then takes these 100 frequency samples and divides them evenly between two GPUs, which contain the real-time beamformer and coarse/fine channel correlator algorithms. Within each HPC is a real-time operating system (RTOS) called HASHPIPE used for thread management and pipelining, and a user interface called dealer/player. These are described in the next sections.

## 3.2 HASHPIPE Implementation

HASHPIPE is an RTOS that pipelines tasks into separate threads with semaphore-controlled shared memory buffers in between each thread [23]. The buffers are circular and consist of the

Figure 3.2: The L-band PAF receiver in the outdoor test facility at GBO. This instrument is placed at the feed of the Green Bank Telescope.

Figure 3.3: The five high performance computers in the Jansky lab at GBO.

number of blocks of data required by the nest thread. Each buffer is controlled by semaphores that signal threads when data is available/unavailable. The semaphore value is set to one when the buffer is filled, and set to zero when it is free. If the block's semaphore is set to zero, any thread that requests the block's data will hang until the semaphore is set to one. The structure of HASHPIPE can be seen in Figure 3.5.

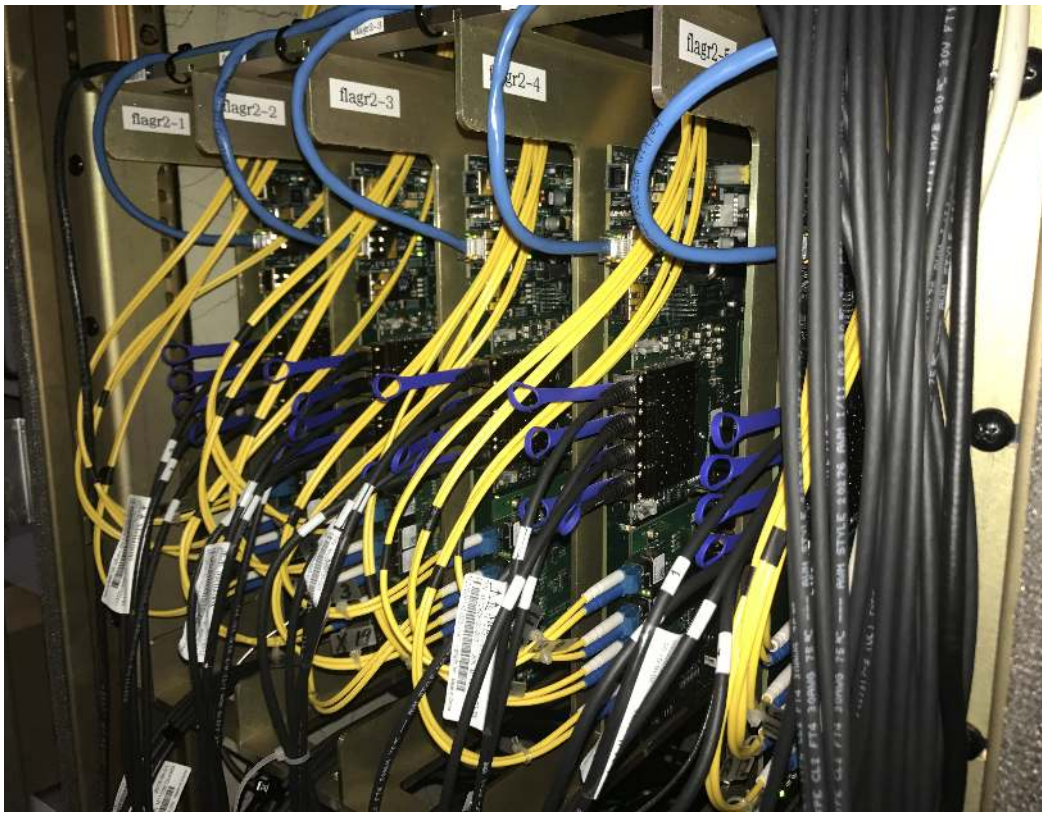



Figure 3.4: The five ROACH II FPGA boards in the Jansky lab at GBO. The four 10 Gbe cables seen on the board combine into a 40 Gbe cable which is connected to the ethernet switch.

Various HASHPIPE threads have a shared object that defines the thread and shared memory buffer parameters. These shared objects are called HASHPIPE plugins and provide the threads with functionality. The plugins use multiple threads to perform a task and align the threads accordingly.

The threads of importance to this thesis are the net, transpose, correlator, and beamform thread. This requires two HASHPIPE plugins; a correlator and beamformer plugin. Both correlator and beamformer plugins comprise of a net, and transpose thread and an operational mode thread (correlator and beamformer respectively). The thread layout can be seen in Figure 3.6.

The net thread is a network sniffing thread which listens on the network for UDP packets from ROACH boards. The transpose thread reformats the data so that the time samples are aligned correctly in memory. And the operatonal mode thread correlates or beamforms the data depending on the chosen mode. The data is then written to flexible image transport (FITS) files using a FITS writer (this was developed by students at West Virginia University) and saved to a Lustre disk array [24].

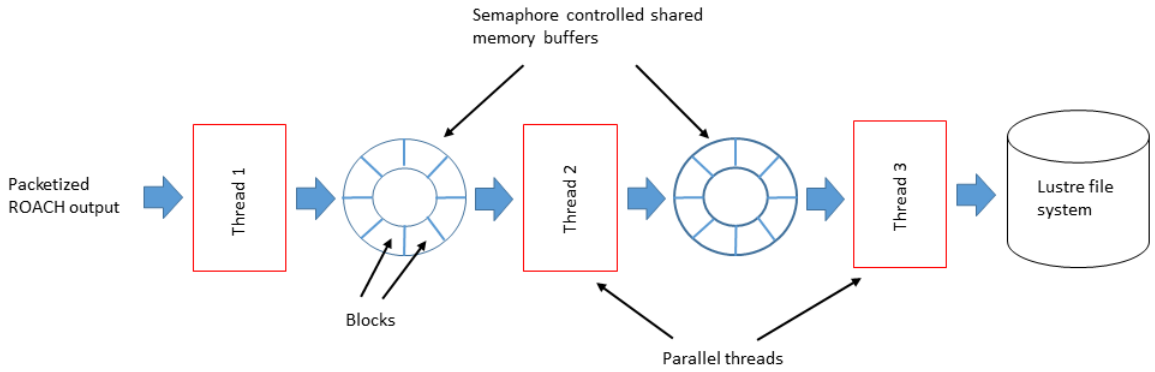



Figure 3.5: An example of the structure of HASHPIPE. Between each thread is a semaphore controlled buffer that transfers data when the buffer is full. After processing the data is stored in the Lustre file system at GBO.

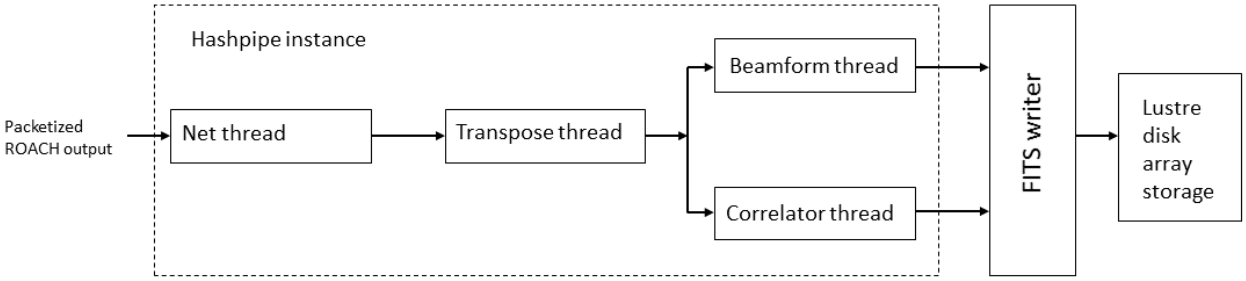



Figure 3.6: A single HASHPIPE instance with a FITS writer and Lustre file system. The net thread captures the data, the transpose thread restructures the data, the beamformer/correlator thread processes the data, and the data is sent to the FITS writer which stores it in FITS files.

## 3.3 Dealer/Player

HASHPIPE is managed by an interface called dealer/player. The dealers and players are part of a distributed system written in Python called the digital backend system (DIBAS) spectrometer control software [25]. The system consists of back end servers, each running on a HPC, designated players.

There are four players per HPC and each player manages an instance of HASHPIPE code and FITS writer. Each player/HASHPIPE instance processes 25 frequency bins of data. The players can be run interactively or as daemons (as a background process that may remain running indefinitely).

The players are controlled and coordinated by a simple client library called the dealer. It may be run interactively or may be imported into scripts.

The dealer/player interface is controlled by a configuration file which outlines the parameters for each player and the various operational modes that the player can run. Most of this thesis focuses on the beamformer operational mode which is described in the following sections.

## 3.4 Beamforming

When generating beamformer output for FLAG, an additional dimension of frequency, is considered. The beamformer output is given by

$$b_{k,j}[n] = \mathbf{w}_{k,j}^{H}\mathbf{x}[n], \tag{3.1}$$

where $k$ is the coarse channel index, and $j$ is the beam index.

The beamformer output is then reduced (or integrated) in order to obtain sample correlations or integrated spectra per short time integration (STI) window. The integrated spectra is given by

$$S_{k,j}^{c} = \frac{1}{N}\sum_{n=0}^{N-1} b_{k,j}[n]b_{k,j}^{*}[n], \tag{3.2}$$

where $N$ is the number of samples. A block diagram of the beamformer and integrator is shown in Figure 3.7.

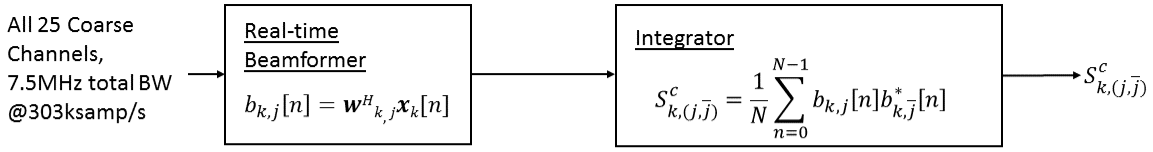



Figure 3.7: Block diagram of the beamformer and integrator at a frequency bin index, $k$, and polarization, $j$.

For beam index $j$, $1 \leq j \leq J$ denotes x polarized beams, and $J+1 \leq j \leq 2J$ denotes the corresponding y polarized beams where $J = 7$ dual polarization beams. Integrated spectra, $S_{k,(j,\bar{j})}^{c}$,

include both self ($xx^*, yy^*$) and cross ($xy^*$) polarized terms, and all three are stored. For self polarized spectra, $\bar{j} = j$ and for cross polarized spectra, $\bar{j} = j \pm J$.

The data in the FLAG digital back end is distributed across five high performance computers (HPCs) at a sample rate of approximately 303 KHz. Each HPC processes 50 coarse channels with a total bandwidth of 15 MHz. There are two GPUs within a HPC that process half of the 15 MHz bandwidth each (7.5 MHz, 25 coarse channels). The following sections discuss the work done to achieve real-time beamforming with these HPCs.

### 3.5 Specifications for Beamformer

The specifications for the beamformer were as follows; a data vector with 4000 time samples, 38 elements and 25 frequency bins, and a weight vector with seven dual polarization beams, 38 elements, and 25 frequency bins. These 25 frequency bins are not contiguous due to the requirements of the fine channel correlator/polyphase filter bank (PFB). This correlator produces 160 fine channels from five of the 25 frequency channels. So the 25 frequency bins are split across the total 500 bins in chunks of 5. Details on the PFB can be seen in [26]. Figure 3.8 shows the structure of these frequency bins at each HASHPIPE instance.

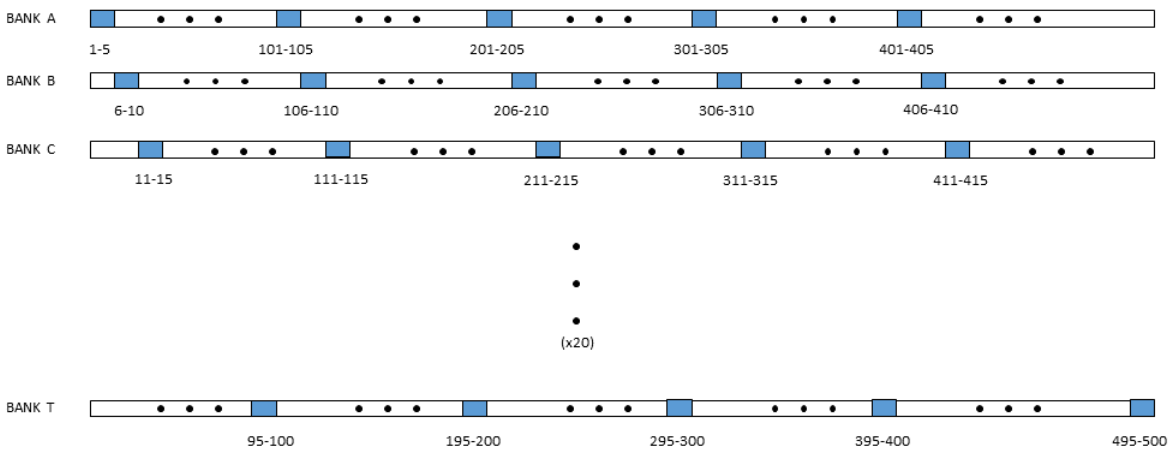



Figure 3.8: Structure of frequency bins at each HASHPIPE instance. This figure shows the non-contiguous frequency bins at each instance. There are sets of five every 100 bins.

Each bank represents the 25 frequency bins processed by each HASHPIPE instance. Since there are four instances per HPC, it follows that there are four banks per HPC.

After beamforming, the beamformer output would then need to be reduced (or integrated) in order to obtain sample correlations or integrated spectra per short time integration (STI) window of which there are 100 with 40 samples each. These values are adjustable, but were otherwise chosen because they allow real-time to be achieved and they are divisible by the 4000 time samples specified.

## 3.6 Beamformer Implementation

The beamforming algorithm was written in three different programming languages, MATLAB, C and CUDA. The algorithm was written in MATLAB and C first, in order to determine the approach for writing it in CUDA.

### 3.6.1 MATLAB Implementation

MATLAB was used to understand the algorithm concepts of beamforming and the mathematics behind it. Initially, the specifications for the beamformer were set as a data vector with 1000 time samples, 38 elements and 25 frequency bins, and a weight vector with seven beams, 38 elements, and 25 frequency bins.

The data and weight array elements were all set equal to one in order to easily identify when the beamformer was working correctly. This method was used again when testing the beamformer in C++, and in CUDA. After verifying that the beamformer worked in MATLAB and understanding the mathematics behind it, code was written in C.

### 3.6.2 C Implementaion

The same specifications were used when writing code in C. This time, however, simulated data and weights were employed in order to further verify correct output, and get closer to achieving the goal of a real-time beamformer. The data and weights contained complex values as they should. An unconventional, but effective method of converting the data and weights from chars to

complex values was used. The real and imaginary components were interleaved in the vectors and were converted to complex vectors by

```
complex[k] = data[2i] + 1j*data[2i+1],
```

where `k` is the complex vector index, and `i` is the char vector index. These complex vectors were represented in rectangular coordinates rather than polar coordinates.

The beamformer algorithm was written in C by using a common method of matrix multiplication using for loops. Since there are three dimensions per array, nested for loops were used.

Another for loop was used to reduce/integrate 25 STI windows with 40 samples each. The result of these reductions were loaded into MATLAB and pre-written code generated plots of two different frequency bins which showed the time average power across seven beams. Examples of the plots generated can be seen in the next section.

While the beamformer in C produced correct results, it was too slow to use in practice since the code was serial. To speed up the beamformer computation so that it could run in real-time, parallel programming using GPUs was implemented.

### 3.6.3 GPU Implementation

A parallel computing platform created by NVIDIA, CUDA, was used to parallelize the beamformer algorithm and achieve real-time speed [27]. CUDA is a language built around C++ (C++ with parallelism). The most important concept to understand in CUDA was how the memory works. There are three different kinds of memory; local, shared and global memory. Each thread has access to local memory which is private to that thread. The thread can read and write from local memory. Threads share memory in a block (collection of threads) which is called shared memory. And every thread can read and write to global memory across blocks. The local memory is faster than the shared which is in turn faster than the global memory [27]. A block diagram of the GPU memory model can be seen in Figure 3.9.

Understanding how the memory works is important because the biggest issue faced in GPU coding was indexing. This was more difficult to debug than the same problem in serial programming due to the greater complexity in memory structure. Debugging the indexing definitely took the longest time, but enabled a better understanding of the GPU memory structure.

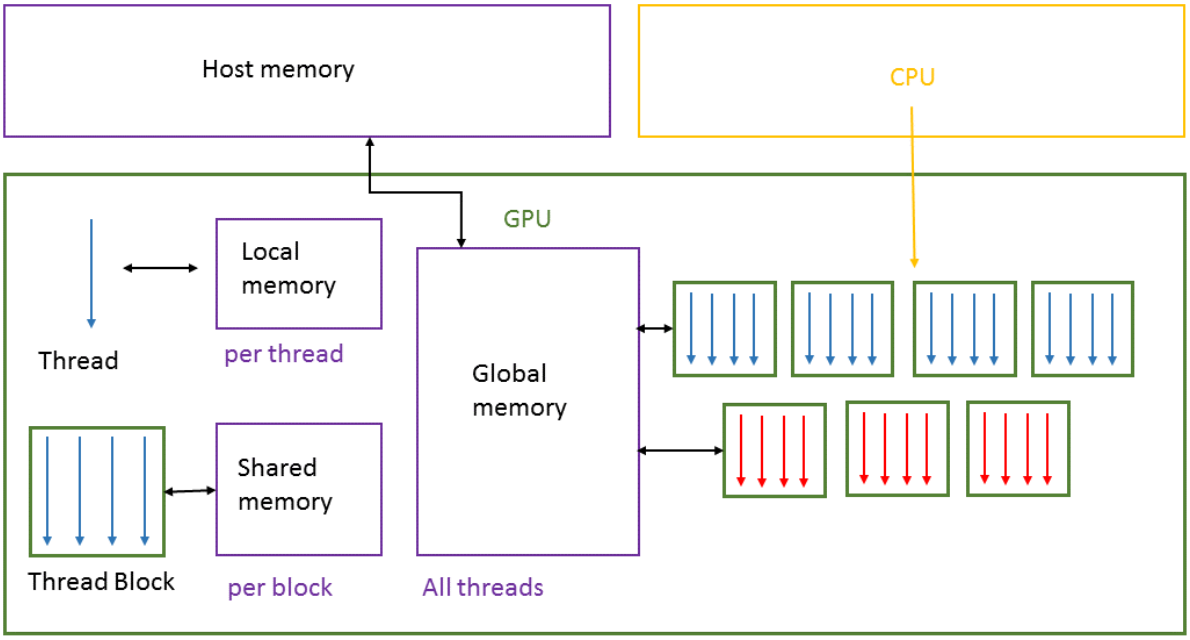



Figure 3.9: GPU memory model showing memory associated with threads. Each thread can read and write to local memory. A block of threads can read and write to shared memory. And every thread can read and write to global memory. The CPU has its own memory which is called host memory in CUDA. Data is copied to and from host memory to global memory on the GPU.

The code was written using the same matrix multiplication method used in C, but parallelized with a reduction algorithm developed for CUDA [27]. The beamformer output was then integrated using the same reduction algorithm.

After debugging this code, the beamformer was capable of real-time (real-time is calculated by dividing the number of time samples by the sample rate) and produced correct results. While this was good, some modifications had to be made to the beamformer due to changes in specifications.

The polarization of the beams changed from single to dual polarization beams making the total number 14 rather than 7, and the final output of the beamformer after integration had to include the self polarized ($xx^*, yy^*$), and cross polarized ($xy^*$, $x^*y$) products. The total number of time samples also changed from 1000 to 4000. After these specifications were met, the beamformer computation time was within three milliseconds of real-time (real-time being 13.2 milliseconds) excluding copies from the CPU to the GPU. The total processing time was higher than real-time so something needed to be done to increase computation speed.

Fortunately, CUDA contains a library called cuBLAS (CUDA Basic Linear Algebra Subroutines) that has functions that can perform simultaneous multiplication of batches (small arrays contained within large arrays) [28]. Batches can also simply be thought of as multiple matrices within arrays. The function needed for beamforming was called `cublasCgemmBatched()` [28]. The format of data and weight arrays being multiplied can be seen in Figure 3.10.

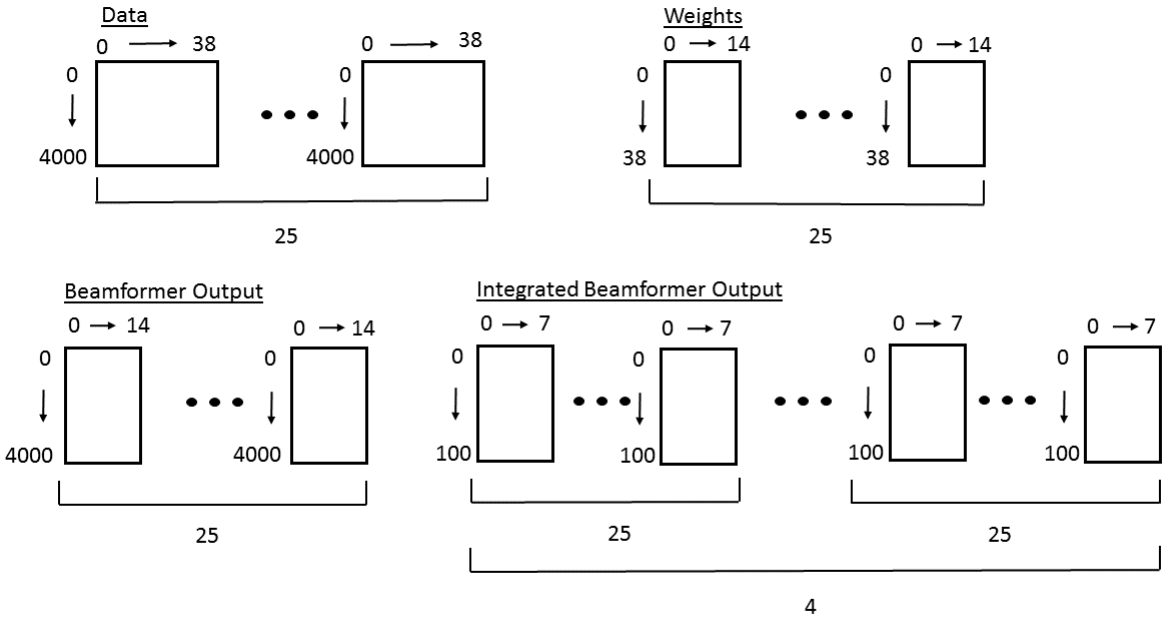



Figure 3.10: Structure of data, weights, `cublasCgemmBatched()` output, and integrated beamformer output. The data had dimensions $4000 \times 38 \times 25$, the weights had $38 \times 14 \times 25$, the dimensions of the output from `cublasCgemmBatched()` were $4000 \times 14 \times 25$, and after integration, $100 \times 7 \times 25 \times 4$. The first dimension is the 100 STI windows, and the last dimension is the 4 polarizations (self and cross polarizations).

The dimensions of the data are: number of time samples by elements by frequency bins ($4000 \times 38 \times 25$), and those of the weights are: number of elements by beams by frequency bins ($38 \times 14 \times 25$). This function multiplies 25 matrices of data and weights together (the 25 matrices represent 25 frequency bins per HASHPIPE instance) to produce 25 matrices of beamformer output. The beamformer output is then integrated with the same reduction algorithm as before.

After a little debugging, the beamformer was capable of achieving approximately 843 microseconds in computation time. Including time taken to copy memory from the CPU to GPU

and vice versa, the total time was approximately five milliseconds which is well below the 13.2 milliseconds needed to achieve real-time operation.

### 3.7 Simulated Data and Weights

In order to test the beamformer, simulated data and weights were generated in MATLAB. The data was modeled as follows:

$$\mathbf{x}[n] = \mathbf{a}(\theta)s[n], \tag{3.3}$$

where $s[n]$ is the signal of interest (SOI), and $\mathbf{a}(\theta)$ is a normalized array response to a unit amplitude point source in the far field at directions corresponding to $s[n]$. Noise and interference were not generated for the initial tests out of convenience, but will be added later.

The elements of the steering vector $\mathbf{a}(\theta)$ are complex exponentials of the form $e^{j\phi}$. The steering vector can be expressed as

$$\mathbf{a}(\theta) = \begin{bmatrix} 1 \\ e^{j\phi_1(\tau)} \\ e^{j\phi_2(\tau)} \\ \vdots \\ e^{j\phi_m(\tau)} \end{bmatrix} \tag{3.4}$$

where

$$\phi_m(\tau) = 2\pi f m \tau, \tag{3.5}$$

$$\tau = \frac{d\cos(\theta)}{c}, \tag{3.6}$$

$m$ represents the element index of the aperture array, $f$ is the frequency bin index, $\tau$ is the time delay due to propagation across the elements, $d$ is the distance between elements, $\theta$ is the angle of arrival corresponding to the signal of interest, and $c$ is the speed of light. The equation to calculate

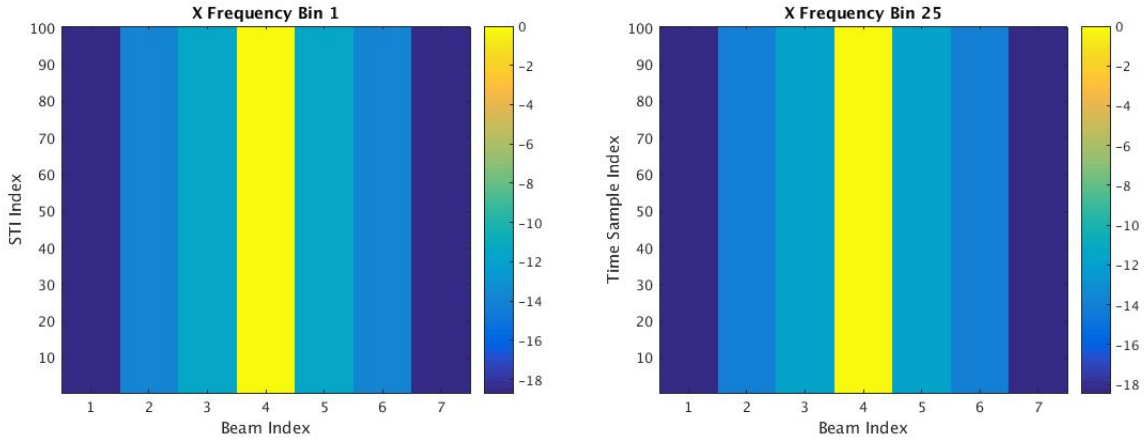
the phase, $\phi_m(\tau)$, is used under the assumption that the frequency is narrow band. For testing purposes the data was generated for a linear aperture array.

### 3.8 Testing

The first test was very simple, the steering vector, $\mathbf{a}(\theta)$, and the weight vector, $\mathbf{w}$, were equal with the signal of interest being set to a value of one at all time samples. The output should have had the same value across all beams and time samples which it did. If the weights and the steering vector have the same values then the conjugate of the weights multiplied by the steering vector and signal amplitude would result in a vector of ones. This is given by

$$\mathbf{w}^H\mathbf{a}(\theta)s[n] = \begin{bmatrix} 1 & e^{-j\phi_1} & e^{-j\phi_2} & \dots & e^{-j\phi_m} \end{bmatrix} \begin{bmatrix} 1 \\ e^{j\phi_1} \\ e^{j\phi_2} \\ \vdots \\ e^{j\phi_m} \end{bmatrix} s[n]. \tag{3.7}$$

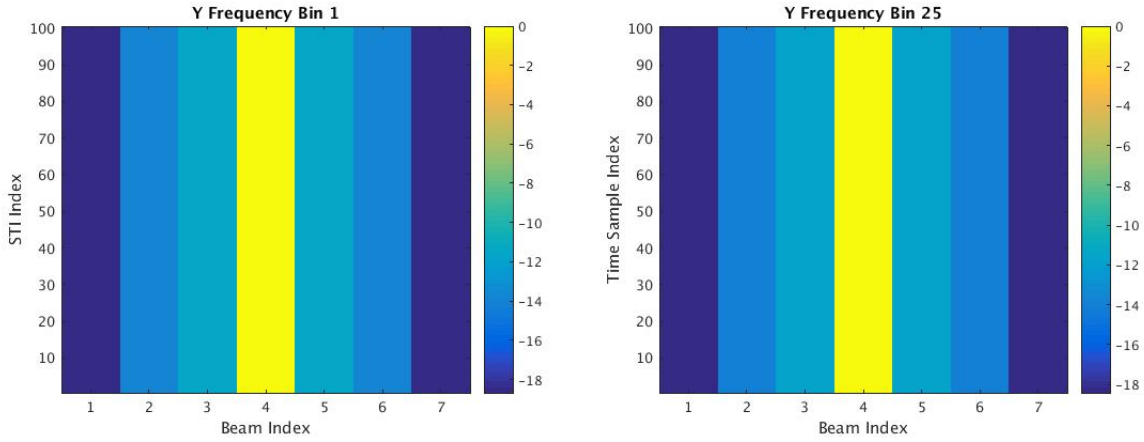
The second test was also fairly simple with the signal of interest in the direction of the main lobe of one of the beams. The expected beam pattern was observed at the output. This can be seen in Figures 3.11 through 3.14, which show the self polarization and cross polarization beams.




(a) Frequency bin 1

(b) Frequency bin 25

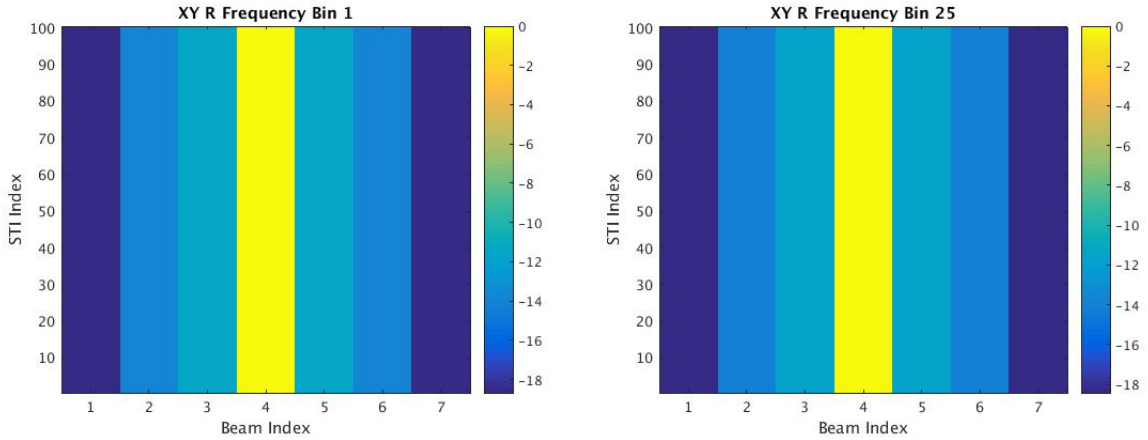
Figure 3.11: X polarized beams with a stationary signal of interest at frequency bin 1 and 25. The signal was placed in the center beam of seven with 100 STI windows. The power is normalized to unity.




(a) Frequency bin 1

(b) Frequency bin 25

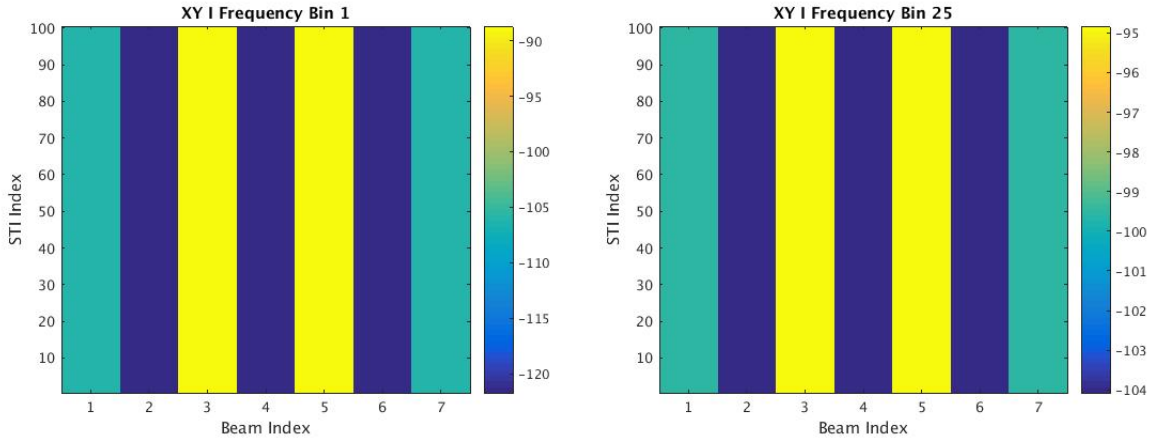
Figure 3.12: Y polarized beams with a stationary signal of interest at frequency bin 1 and 25. The signal was placed in the center beam of seven with 100 STI windows. The power is normalized to unity.




(a) Frequency bin 1

(b) Frequency bin 25

Figure 3.13: Real cross-polarized beams with a stationary signal of interest at frequency bin 1 and 25. The signal was placed in the center beam of seven with 100 STI windows. The power is normalized to unity.

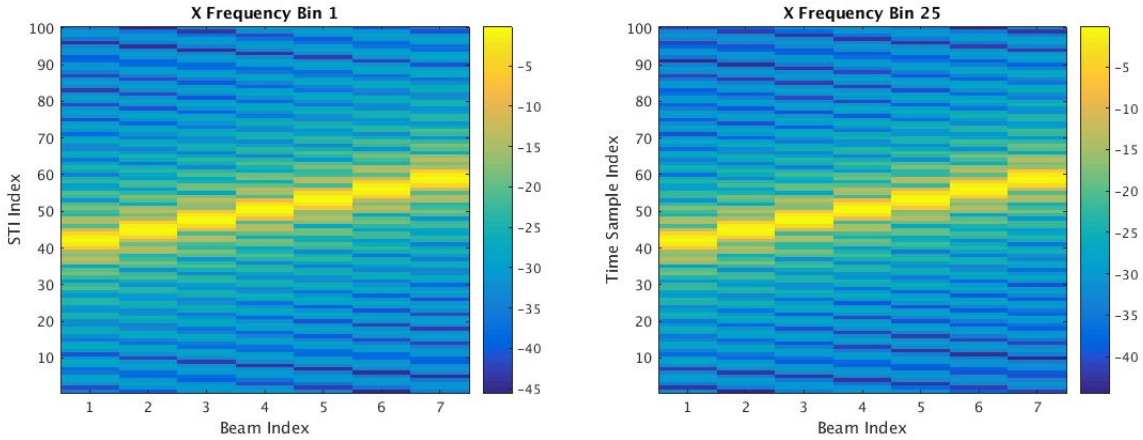



(a) Frequency bin 1

(b) Frequency bin 25

Figure 3.14: Imaginary cross-polarized beams with a stationary signal of interest at frequency bin 1 and 25. The signal was placed in the center beam of seven with 100 STI windows. The power is normalized to unity.

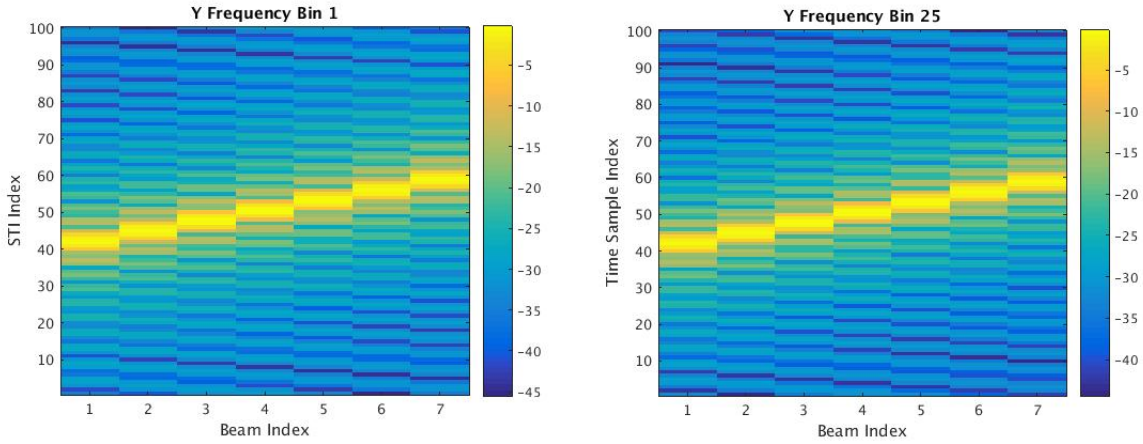
The third test had the signal of interest moving across all the beams which can be seen below. The signal of interest is clearly seen moving across the beams.




(a) Frequency bin 1

(b) Frequency bin 25

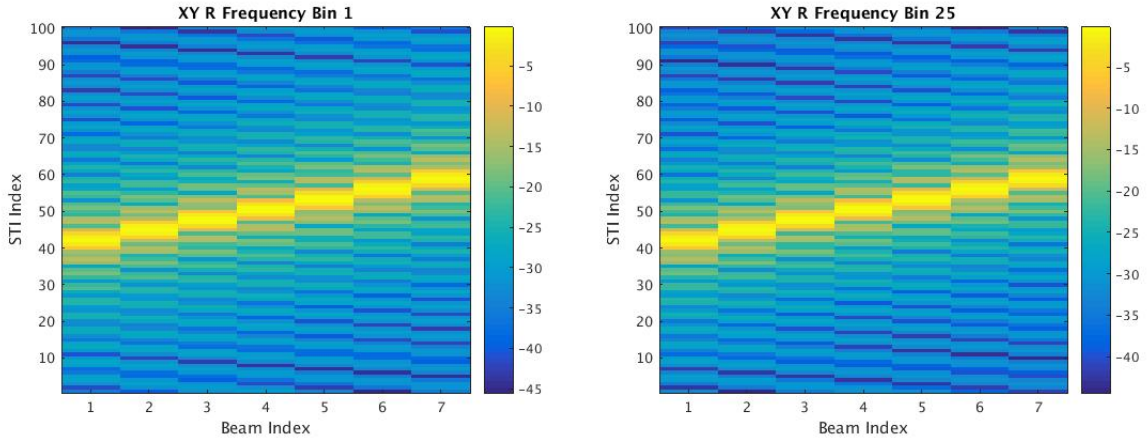
Figure 3.15: X polarized beams with a moving signal of interest at frequency bin 1 and 25. The signal moved across the seven beams, and 100 STI windows. The power is normalized to unity.




(a) Frequency bin 1

(b) Frequency bin 25

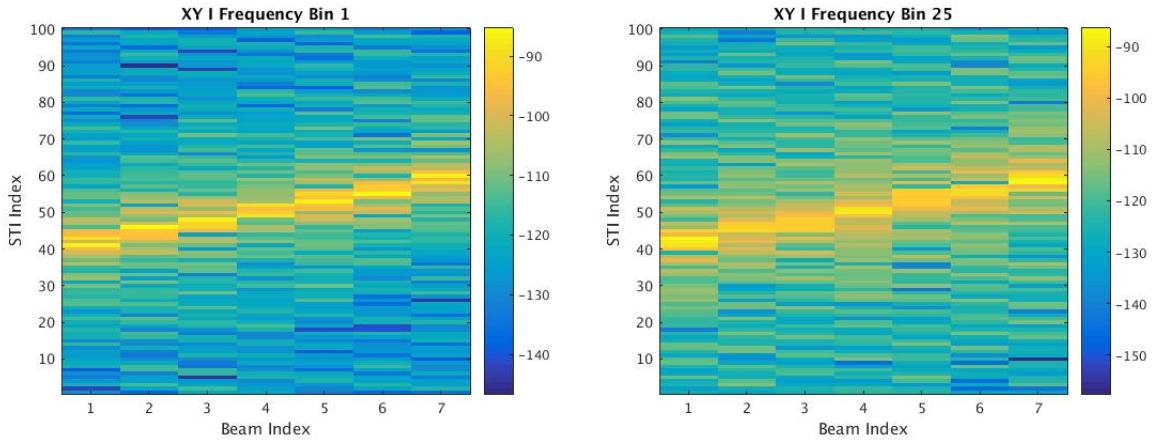
Figure 3.16: Y polarized beams with a moving signal of interest at frequency bin 1 and 25. The signal moved across the seven beams, and 100 STI windows. The power is normalized to unity.




(a) Frequency bin 1

(b) Frequency bin 25

Figure 3.17: Real cross polarized beams with a moving signal of interest at frequency bin 1 and 25. The signal moved across the seven beams, and 100 STI windows. The power is normalized to unity.

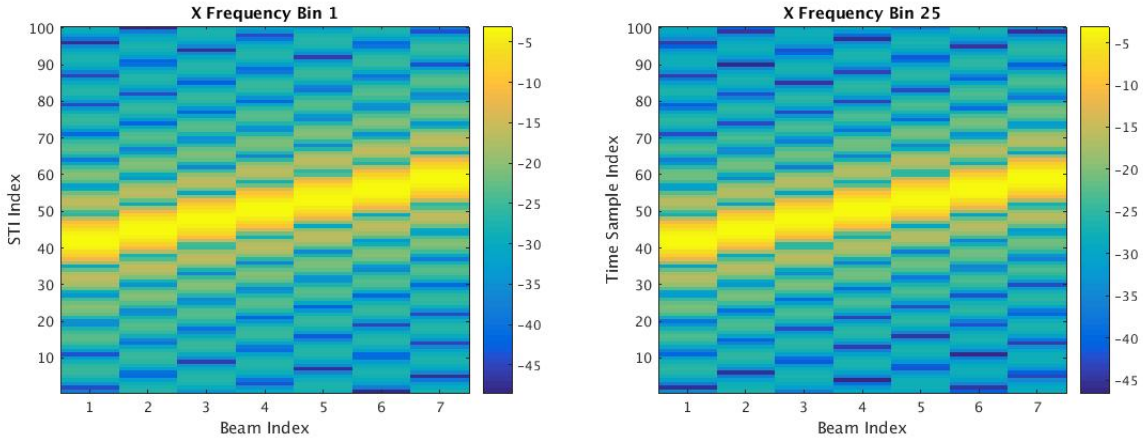



(a) Frequency bin 1

(b) Frequency bin 25

Figure 3.18: Imaginary cross polarized beams with a moving signal of interest at frequency bin 1 and 25. The signal moved across the seven beams, and 100 STI windows. The power is normalized to unity.

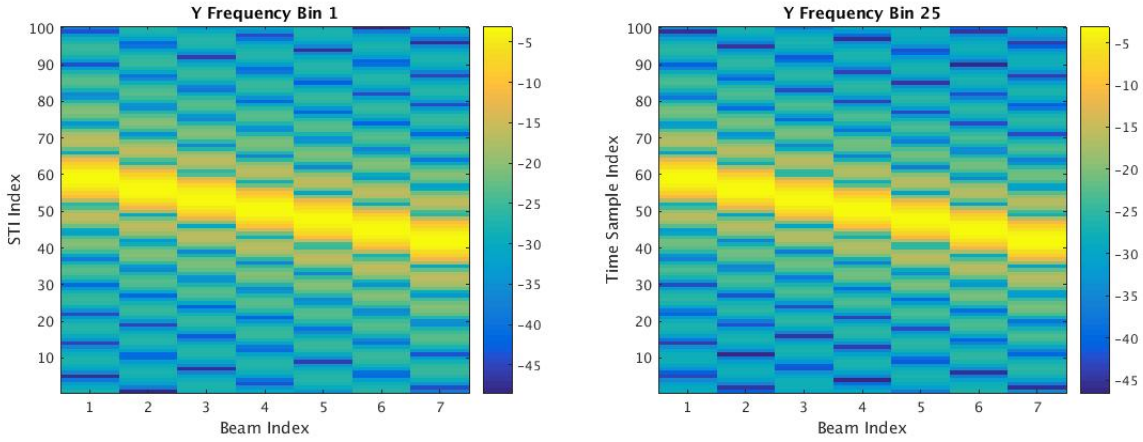
The previous tests were all done to test the self polarizations ($XX^*, YY^*$). The last test was done for the cross polarization ($XY^*$) with a signal of interest moving across the X polarized beams in the opposite direction of the signal of interest moving across the Y polarized beams.




(a) Frequency bin 1

(b) Frequency bin 25

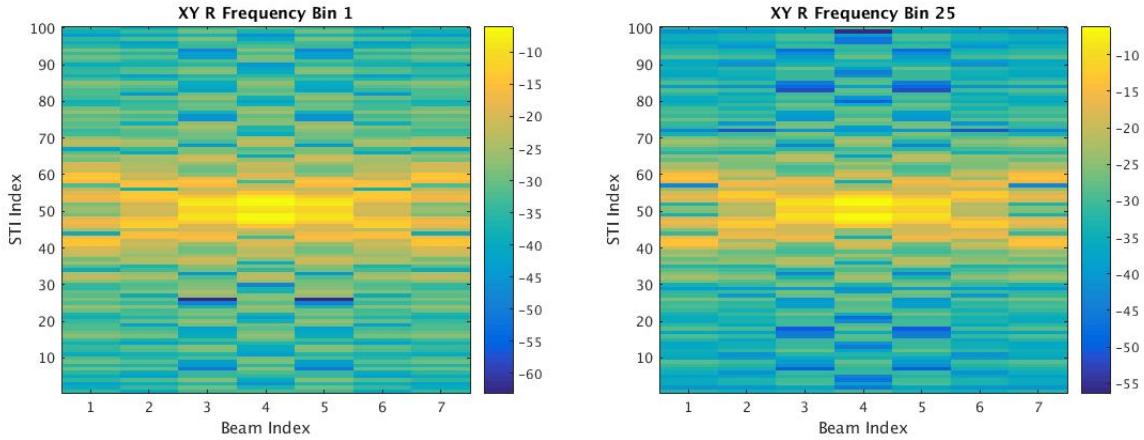
Figure 3.19: X polarized beams with a moving signal of interest opposite to that of the Y polarized beams at frequency bin 1 and 25. The signal moved across the seven beams, and 100 STI windows. The power is normalized to unity.




(a) Frequency bin 1

(b) Frequency bin 25

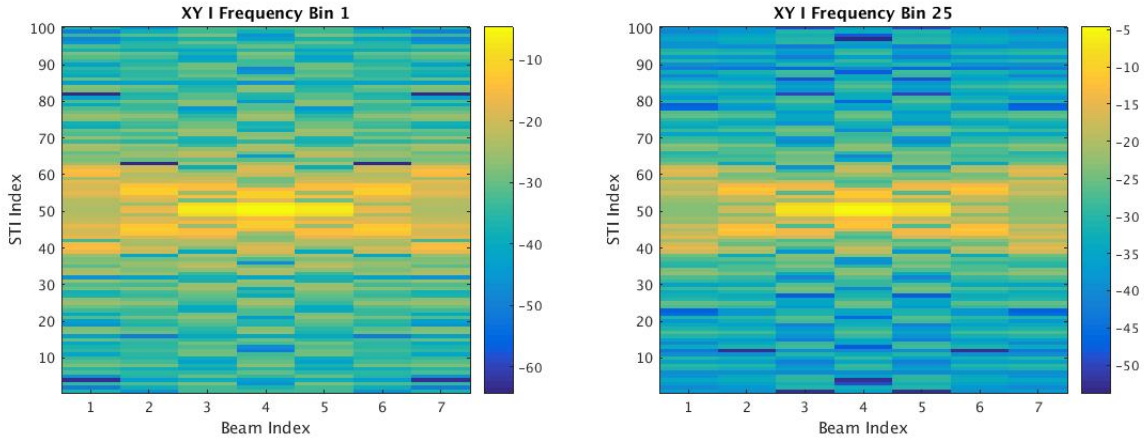
Figure 3.20: Y polarized beams with a moving signal of interest opposite to that of the X polarized beams at frequency bin 1 and 25. The signal moved across the seven beams, and 100 STI windows. The power is normalized to unity.




(a) Frequency bin 1

(b) Frequency bin 25

Figure 3.21: Real cross polarized beams with a signal of interest moving in opposite directions corresponding to X and Y polarized beams at frequency bin 1 and 25. The signal moved across the seven beams, and 100 STI windows. The power is normalized to unity.




(a) Frequency bin 1

(b) Frequency bin 25

Figure 3.22: Imaginary cross polarized beams with a signal of interest moving in opposite directions corresponding to X and Y polarized beams at frequency bin 1 and 25. The signal moved across the seven beams, and 100 STI windows. The power is normalized to unity.

### 3.9 Real-time Beamformer Test

In order to verify that the real-time beamformer output was correct after integrating the transpose thread into the CUDA code (discussed in a later chapter), simulated pulsar data was generated using MATLAB. This data was packetized with MATLAB and sent to the input ports of the HPC to simulate data acquisition. To simulate the pulsar, the time delay, $t_2 - t_1$, between two observing frequencies (two frequency bins), $f_1$ and $f_2$, was used. Let $\tau_k = t_{k+1} - t_k$,

$$\tau_k = 4.1488 \times 10^{-3}(f_k^{-2} - f_{k+1}^{-2})\text{DM}, \tag{3.8}$$

where $k$ is the frequency channel index and is within the range 0 to 498, DM is the dispersion measure which is the number of free electrons between the observer and the pulsar per unit area. The larger the DM, the slower the pulsar.

After calculating the time delay, it is then applied to the phase of the pulsar as shown in Equation 3.10,

$$\phi(t) = 2\pi m f_k(t - \tau_k), \tag{3.9}$$

$$p(t) = Ae^{j\phi(t)}, \tag{3.10}$$

where $k$ is the frequency channel index and is within the range 0 to 499, $m$ is the element index, and $A$ is the signal amplitude. A simulation can then be generated and shown with a time vs frequency plot as shown in Figure 3.23. An additional plot is provided showing the pulsar change in a lower number of frequency bins and time samples.

In order to compare results with the real-time beamformer, additional code was written to simulate the beamformed output. This was done using a weight vector whose elements were all equal to one and using the same weights when running the real-time beamformer.

Both sets of data had a dispersion measure of 10. The output of the real-time beamformer appears to be more precise than that of the simulated MATLAB output. This is probably due to higher precision in the real-time beamformer compared to the simulated MATLAB output. From this test, it can be seen that the real-time beamformer generates correct output and can be used for detection of radio transients.

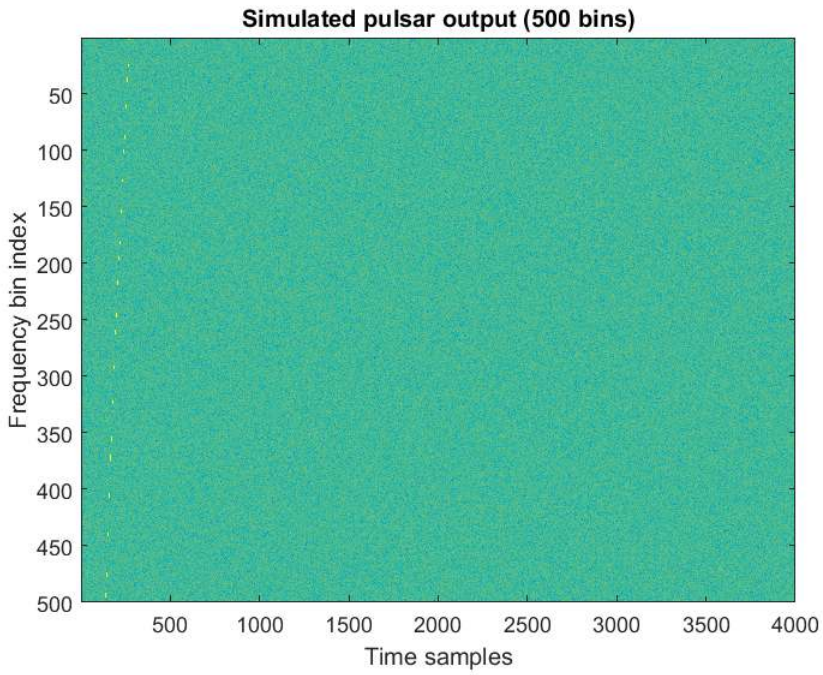



Figure 3.23: A plot of the simulated pulsar as a function of frequency with a dispersion measure of 10.

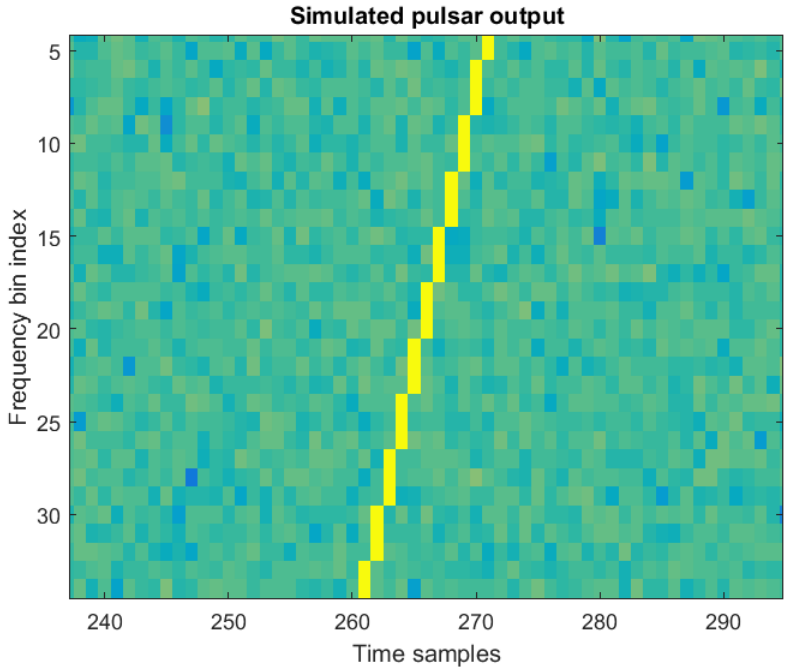
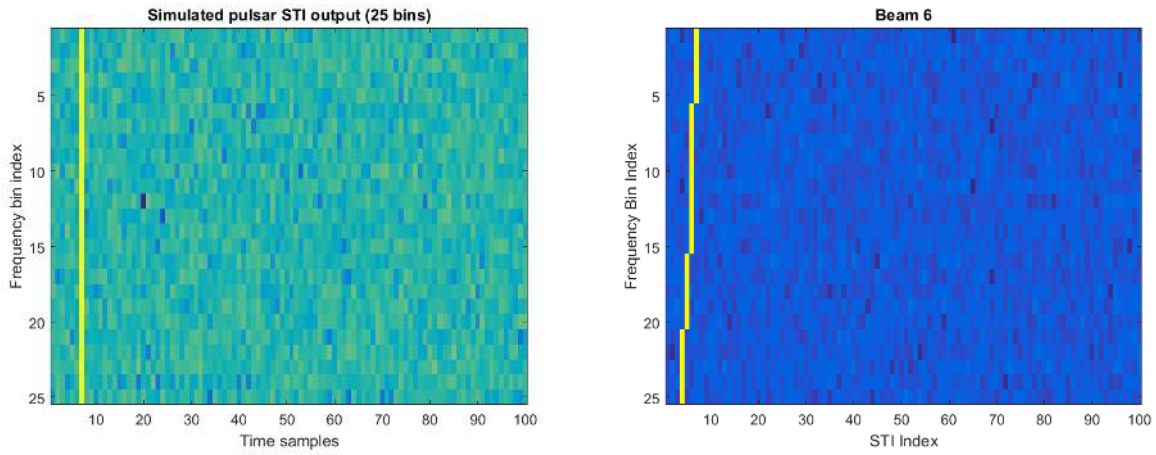



Figure 3.24: Plot of the simulated pulsar in Figure 3.23 showing a clearer change in frequency over time by focusing on a smaller number of frequency bins and time samples.




(a) Simulated in MATLAB

(b) RTBF output

Figure 3.25: Beamformer output generated in MATLAB with the same dimensions of data and weights in (a). Real-time beamformer output using the same simulated pulsar data in (b).

## 3.10 Summary

Real-time beamforming was achieved with the help of the cuBLAS library, and with a few changes made to the code. The time taken for beamforming, and short time integration is approximately 843$\mu$s with the total duration of time being approximately 5 ms (this includes memory allocation, as well as copies from the CPU to GPU and vice versa). The beamformer CUDA code was then integrated into HASHPIPE with the same results using the same simulated data and weights.

During one of the live tests of the system at GBO, the data restructure in the transpose thread of the beamformer HASHPIPE code was integrated into the CUDA code. This resulted in the removal of the beamformer transpose thread which was tested with simulated pulsar data. The following chapters provide details on the live tests of the FLAG back end as well as modifications made between tests.

# CHAPTER 4. JULY 2016 FLAG COMMISSIONING

## 4.1 Introduction

The following chapters provides details on the FLAG back end tests carried out at the Green Bank observatory. Each of these tests was called a commissioning, and involved several hours of data acquisition split over two to three weeks. A continuous set of hours spent acquiring data was called a session, and was spent testing the operational modes of the back end. The following sections describe each of the commissioning runs and show results obtained.

In this chapter, the work done, and results acquired during the July 2016 commissioning of FLAG are discussed. A few details of the digital back end are addressed in this chapter and more will be discussed later on.

As mentioned previously, there are four players per HPC and five HPCs total. The HPCs are designated FLAG one to five and only two of the five HPCs were used during this commissioning, FLAG two and three. The players can either be run interactively or as daemons. They were run as daemons during the entire commissioning. The players are controlled and coordinated by a simple client library called the dealer. It may also be run interactively or may be imported into scripts.

Both methods were used for different reasons. The dealer was run interactively in order to debug or change certain parameters, such as the integration length, faster and it was run from a script when no debugging or change of parameters were needed (usually during a scan which refers to the telescope moving towards, moving away or dwelling on a source). During observation times, all that needed to be done was starting the dealer and players. The players were run as daemons and the dealer waited for a scan to start.

The first test that we ran was called the simulator 3 test (the third of simulator tests run on the back end). This test was done to make sure that the back end was working as expected. Simulated data was stored on the binary random access memory (BRAM) of the ROACH II FPGA [21], and was used to test the back end of the system. All that needed to be done was a simple

change in a configuration file to ensure that data from the BRAM was used rather than data from an actual scan.

### 4.2 Bit/Byte lock

After we verified that simulator 3 worked, the next step was to incorporate the front end and test the full system. This test was basically the same on our end, start the players and dealer and wait for data. But before a scan was started, the data needed to be bit and byte locked [29]. Bit locking is a process that ensures that the first bit of the inphase or quadrature 8-bit word received is indeed the first bit of that 8-bit word. Byte locking is a process that makes the distinction between inphase and quadrature (which eight bits are the inphase component and which ones are the quadrature component).

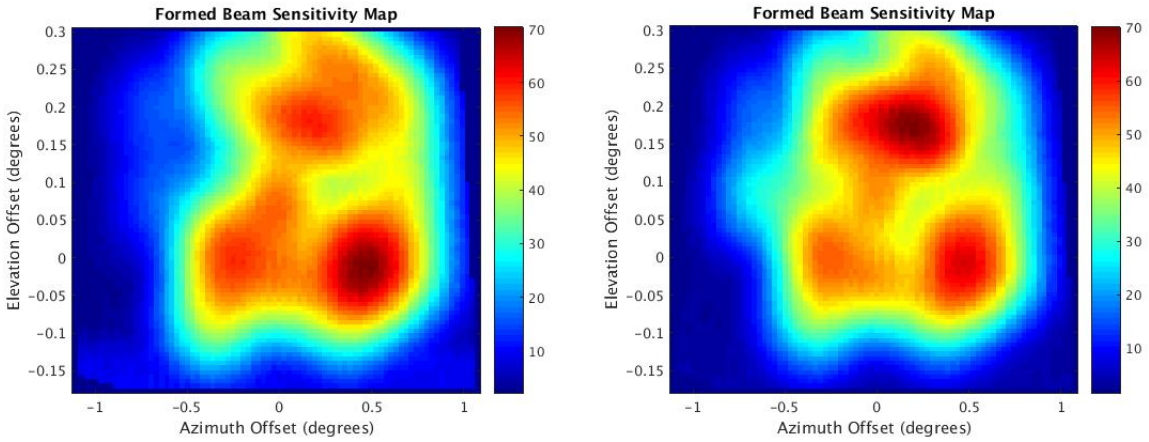
After the bit and byte locking process was complete, the scan was started. Initially, we were loosing a lot of packets of data or all of the data using one or two players on FLAG three. This was because a few of the packets would come in late which led to the rest of the packets being seen as late. This was fixed by advancing to the next m-count (one m-count consists of five packets) despite having a few late packets at the beginning of the scan. 10 to 15 minute long scans were run after this bug was fixed and we were able to acquire quite a bit of data despite our short observation windows. FLAG two was added to increase the amount of bandwidth that we were scanning over. Each player acquired data at a bandwidth that we scanned.

Since four players were running per HPC, we acquired four times the bandwidth of one player (The bandwidth acquired by one player was 7.5 MHz). On FLAG two, only two players were working so together with FLAG three, we acquired six times the bandwidth of one player ($3/10$ of the total bandwidth).

One other problem was the bits were not locking for some of the X-polarized elements and a few of the Y-polarized elements. This was out of our control and given the limited time we had, we had to deal with it. The front end was also experiencing some heating issues. Despite these problems, we were still able to acquire useful data during our observation windows.

## 4.3 GBT Results

With the data that we acquired, we were able to plot sensitivity maps, beam patterns, and element patterns. Each one of these plots can be seen in the figures below. The x and y axes of the figures are the azimuth and elevation offsets, respectively. This data was acquired from a grid scan of a source called 3C295.

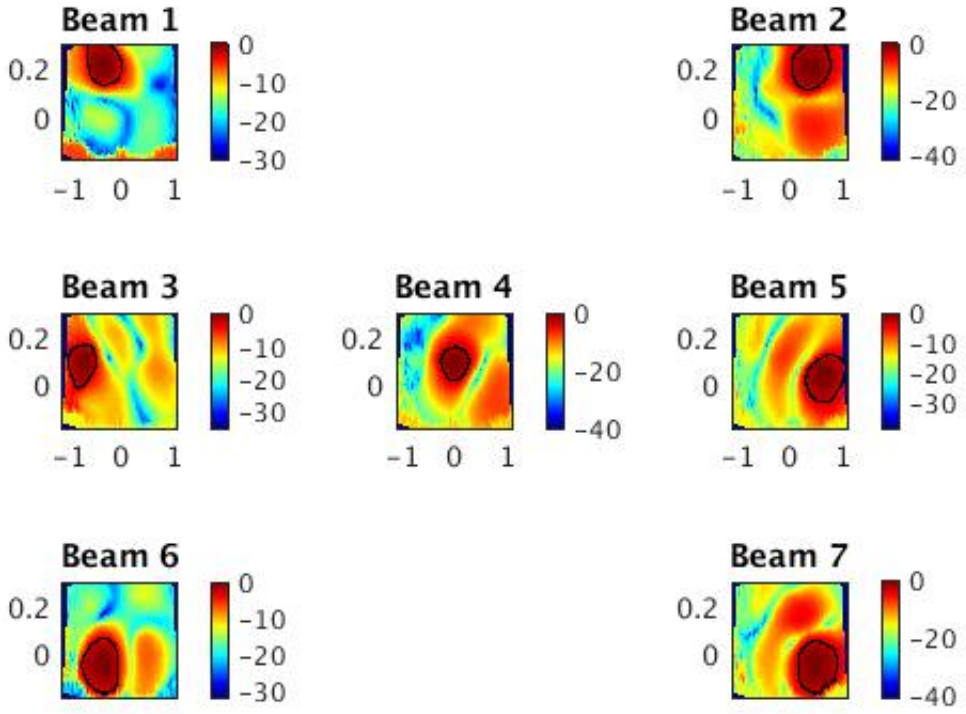



(a) Frequency bin 1

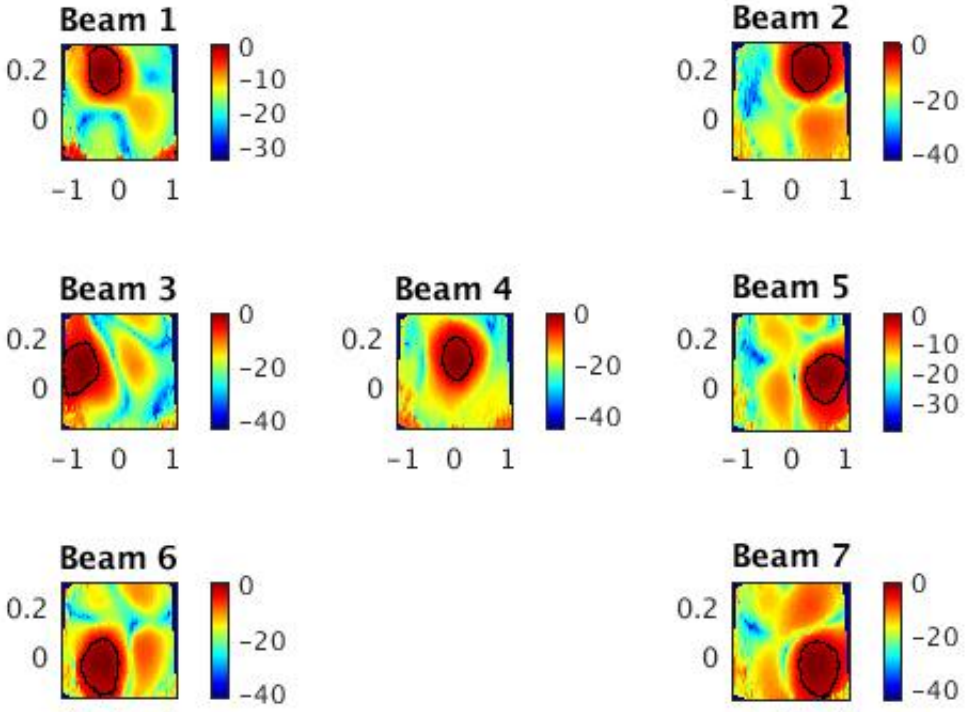
(b) Frequency bin 25

Figure 4.1: Formed beam sensitivity maps of the first and last frequency bin indices (1360 MHz and 1483 MHz respectively).

The maximum sensitivity of 70 dB means that the system temperature ($T_{\text{sys}}$) is high given a high antenna efficiency ($\eta$). At this point, the $T_{\text{sys}}$ is approximately 85 K which is quite high, but significantly better than the 200 K that was calculated earlier. This was troubling since the goal $T_{\text{sys}}/\eta$ was approximately 28 K (a single pixel feed $T_{\text{sys}}/\eta$). Fortunately, we were able to measure much lower $T_{\text{sys}}/\eta$ during the next commissioning due to fixes made to the front end by the Green Bank Observatory. Figures 4.3 and 4.3 show beam and element patterns respectively for the same frequency bins as the sensitivity maps in Figure 4.1.

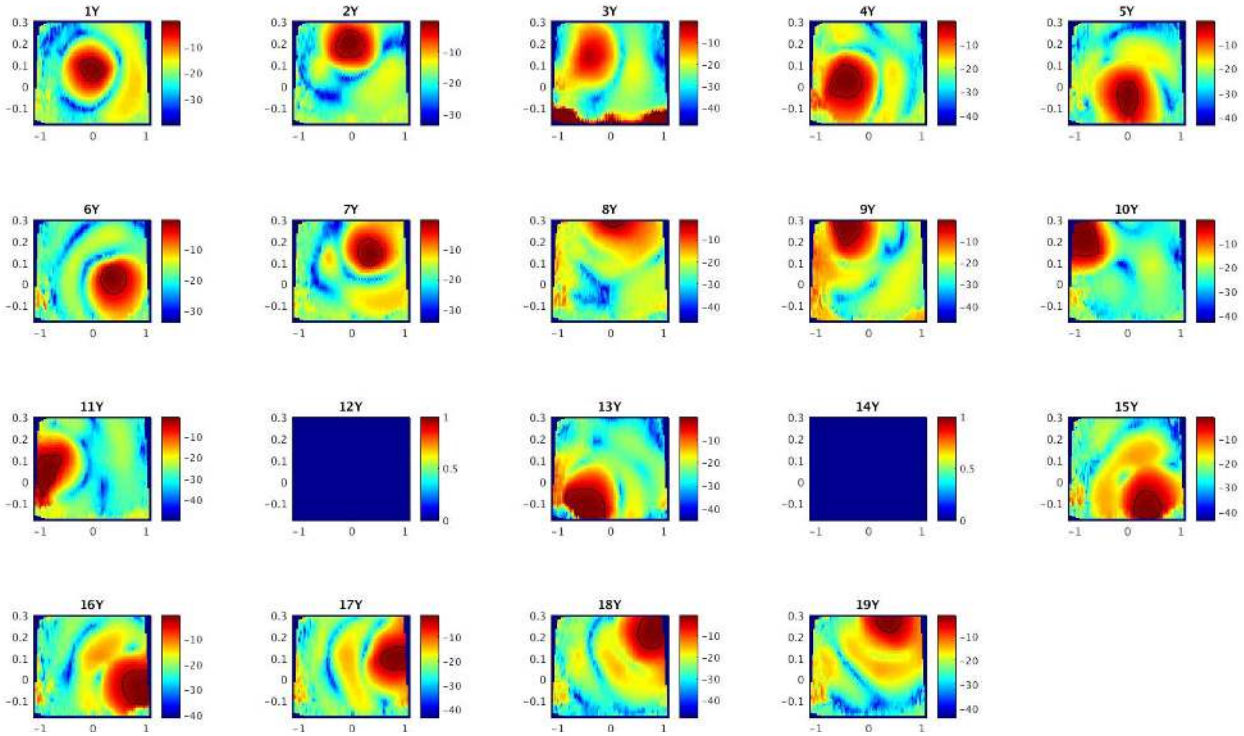



(a) Frequency bin 1

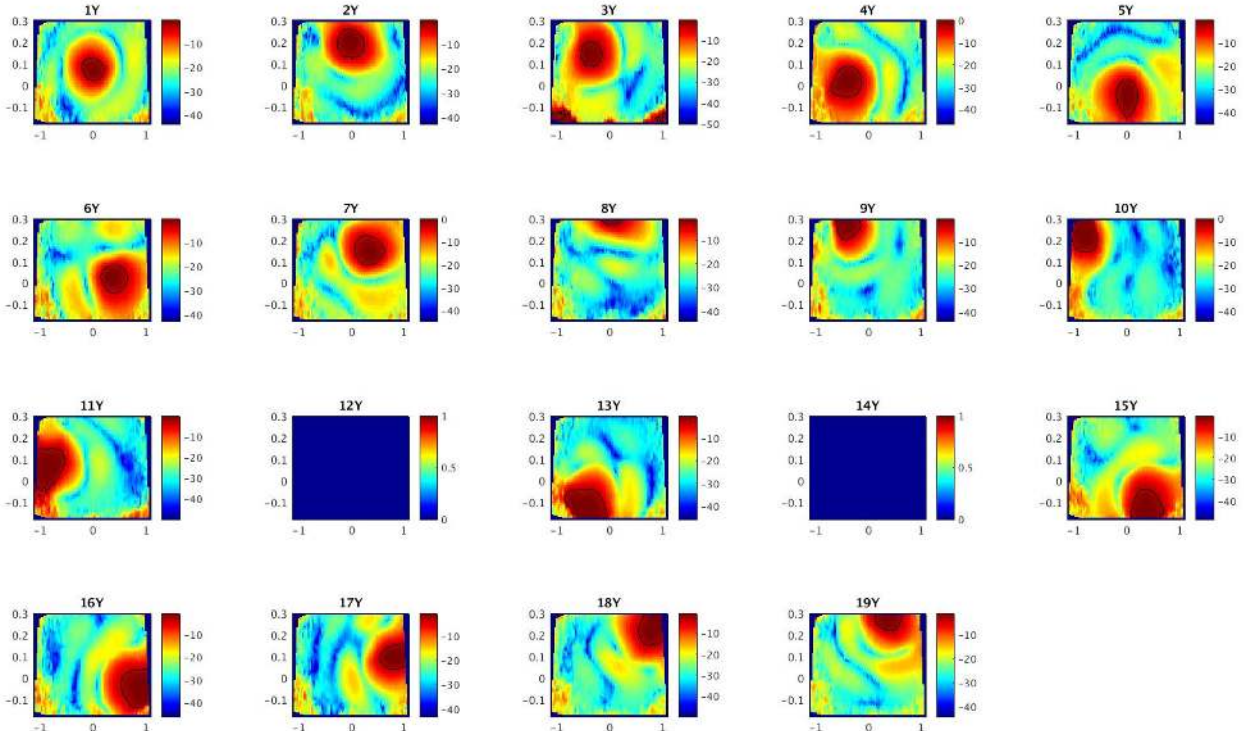



(b) Frequency bin 25

Figure 4.2: Beamformed patterns of the first and last frequency bin indices (1360 MHz and 1483 MHz respectively).




(a) Frequency bin 1




(b) Frequency bin 25

Figure 4.3: Y-polarized element patterns of the first and last frequency bin indices (1360 MHz and 1483 MHz respectively). The blank element patterns are due to non responsive elements.

## 4.4 Summary

During the July 2016 commissioning, despite the fact that we were not able to run the real-time beamformer due to limited time and a few bugs we had to fix in the overall code, we were able to process post-correlated beamformed data. This verified that at the very least, our code was working.

We were also able to to learn the bit/byte locking procedure which ensured the acquisition of accurate data. This is currently being integrated into the back end code to eliminate the time spent using the bit/byte lock GUI.

The $T_{\text{sys}}/\eta$ seen in this commissioning was much higher than expected due to issues in the front end. Fortunately, these were resolved in time for the May 2017 commissioning.

# CHAPTER 5. MAY 2017 FLAG COMMISSIONING

## 5.1 Introduction

In this chapter, the work done, and results acquired during the May 2017 commissioning of FLAG are discussed. More detail is given of the digital back end as well as modifications made since the last commissioning.

During this commissioning, our primary objectives were to acquire data with much lower system temperature ($T_{\text{sys}}$) than the last commissioning, acquire data with the coarse channel correlator, the fine channel correlator, and the real-time beamformer.

Upon my arrival at Green Bank, the PAF was not scheduled to be put on the telescope for a week, so we obtained measurements and tested our code by acquiring data per scan with the PAF in the outdoor test facility (OTF). Data was acquired at night when the temperature was lower because if it exceeded a particular threshold, the system would shut down. The PAF also had to be cooled with ice water that was replaced manually and on a regular basis.

While testing our code with the PAF in the OTF, the priority was acquiring data with low $T_{\text{sys}}$. In order to calculate the $T_{\text{sys}}$, on and off source data needed to be acquired. On source data was acquired with the OTF roof closed, and off source with the roof opened. The roof reliably acted as a deep space source under observation which was used to calculate $T_{\text{sys}}$. Each data acquisition was over a duration of 60 seconds with an integration length of 0.5 seconds which is a dump rate of 1000 samples every 500 milliseconds.

Before acquiring data in the OTF, the data needed to be bit and byte locked (defined in the previous section). Scripts had been written to ensure bit and byte lock before data was acquired, but during the OTF tests, we realized that they were not ready so we resorted to a graphical user interface (GUI) that was used prior to the scripts.

The bit and byte lock procedure, as seen from the GUI, was as follows. Use a generated noise source from the PAF front end for bit lock. Plots were used to determine whether there was

bit lock. Single gaussians meant that the data was bit locked while multiple meant that it was not. Then a test tone was turned on for byte lock. Spectral plots were used to determine byte lock. A tone appearing in either sideband meant that the data was byte locked. This was called sideband separation. Finally, after the data was bit and byte locked, the noise source and test tone were turned off.

After the bit and byte lock procedure, data was acquired using the dealer/player interface addressed in the previous section. A few modifications were made to the interface since the last commissioning which include; a multiple scans feature, code that properly closes FITS files was added, a set weight file parameter, a channel select parameter, and a dealer/player GUI.

## 5.2 DIBAS Modifications

During the last commissioning, we were not able to consecutively acquire data simply by starting another acquisition in the dealer/player interface. The mode and the parameters associated with it had to be reset, and then the acquisition was started which took quite a while and required detailed knowledge of the interface.

We needed to modify the dealer/player code so that we could acquire data consecutively one scan after another. This is what we called the multiple scans feature. The problem was that the code used to write data to FITS files (referred to as the FITS writer) was not enabled by the dealer/player code after other scans were started.

To fix this problem, the stop FITS writer function, located in a file called "Beamformer-backed.py", was overloaded with another stop function that stopped the FITS writer when a scan finished. This allowed it to start when another scan started. Despite having solved the multiple scans issue, another problem arose involving the FITS files. They appeared to be empty while analyzing the data acquired from the OTF during remote testing carried out at BYU after the last commissioning.

The issue was that the files were not being closed properly after the first data acquisition due to a signal handler issue in the FITS writer as well as a memory clean up issue in the HASHPIPE code. These issues were resolved by replacing the signal handler with a stop function rather than an interrupt and adding a condition in the transpose and correlator threads of HASHPIPE, that ensured data was not being processed during memory cleanup.

However, before this was accomplished, I had to manually close the FITS files which required the transfer of the files from one HPC to another. This HPC contained the libraries necessary to close these FITS files. This problem was addressed and fixed before the next commissioning.

After we were confident that data was being acquired and processed correctly with each scan, some modifications needed to be made to the dealer/player interface to make it more user friendly. One of those modifications was a set weight file parameter.

This would enable a user to change the weight file depending on the source position or area being scanned. Code needed to be modified in the dealer/player code as well as the HASHPIPE code. This modification was implemented by creating flags in shared memory that would alert the back end when a weight file was changed, and update the file being used in the beamform thread. These flags were keywords in shared memory written to in the "Beamformerbackend.py" file.

In order to test the functionality, I generated different weight files and run the real-time beamformer, changing the files with each scan. Similar changes were made for a channel select parameter created for the polyphase filter bank (PFB) mode [26].

Another modification that was tested during the latest commissioning, but that is still being worked on, is the dealer/player GUI. This GUI is capable of selecting the players that need to be run, setting the appropriate mode (correlator, real-time beamformer, etc.), setting the integration length, weight files, selecting channels, starting data acquisition and stopping it. The GUI was tested and is only capable of acquiring data given certain conditions and needs to be worked on a little more.

These fixes and modifications led to good results during the May commissioning that will be discussed further in the following sections.

### 5.3 OTF Results

Data was acquired and processed from the OTF and the GBT. As mentioned before, the data acquired in the OTF was used to determine $T_{\text{sys}}$ and to verify that the post-processing code worked. $T_{\text{sys}}$ in the OTF was calculated using the following equation

$$T_{\text{sys}} = \frac{T_{\text{hot}} - T_{\text{cold}} Y}{Y - 1}, \tag{5.1}$$

where $Y$ (the Y factor) is the ratio of on source power (roof closed) to off source power (roof open), and $T_{\text{hot}}$ and $T_{\text{cold}}$ were approximated to be 7.5 K and 290 K respectively. The on and off source power were calculated by taking the real part of the diagonal elements of the corresponding covariance matrices.

The $T_{\text{sys}}$ was calculated over 475 frequency channels rather than 500 since one the HPCs (FLAG five) was having trouble acquiring data due to hardware problems. A plot of the initial $T_{\text{sys}}$ result can be seen in Figure 5.1 where each curve represents a single element.

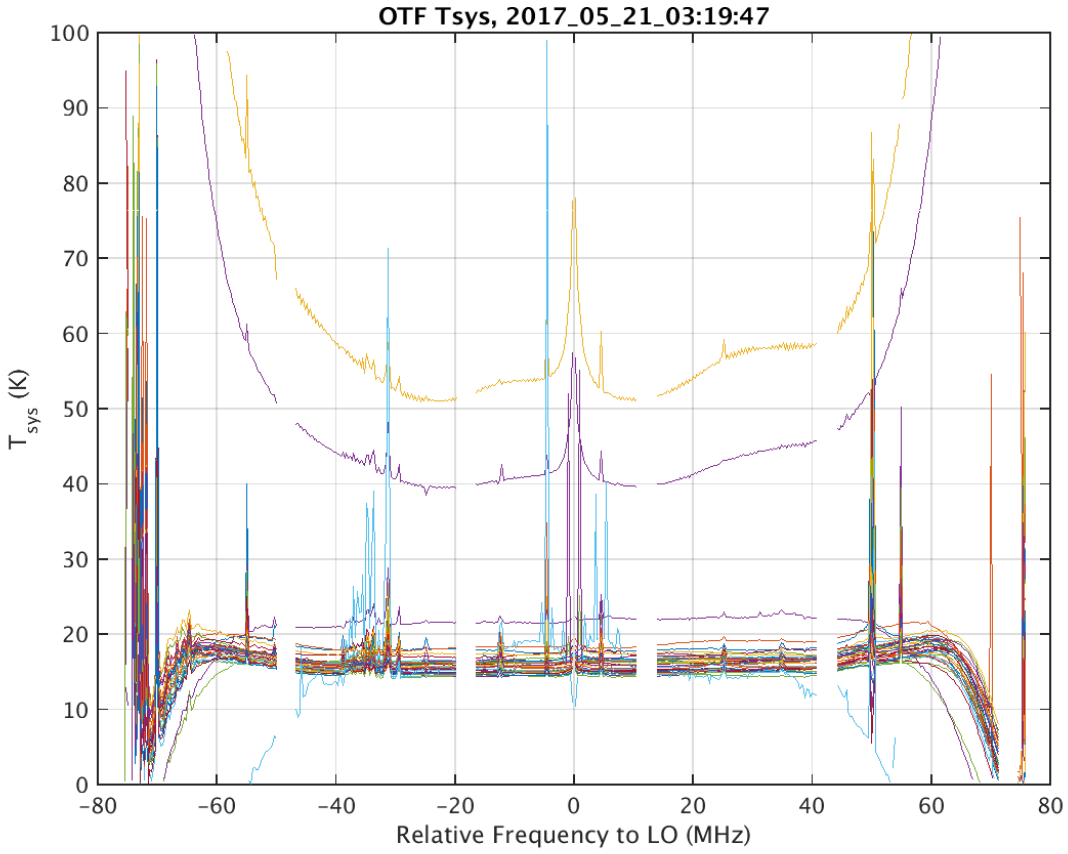



Figure 5.1: $T_{\text{sys}}$ (K) vs. Relative LO frequency (MHz). Each curve represents the $T_{\text{sys}}$ of a single element. The quantization gain was set to 10 and shows a decrease in the transition bands which is inaccurate.

This was suspicious since the $T_{\text{sys}}$ was improving in the transition band. The $T_{\text{sys}}$ should increase as it gets closer to the edges of the band because the signal is attenuated in the transition

band. If the signal is attenuated, the sensitivity decreases as well which leads to an increase in $T_{sys}$ due to the inverse proportionality.

We theorized that the re-quantization of the data in the ROACH was causing our off source measurements to be abnormally low, which made our Y factors high and $T_{sys}$ unusually low. In order to account for this re-quantization, the quantization gain, which was set to 10 in Figure 5.1, needed to be changed.

At the output of the ADC in the ROACH and after the sideband separation step for bit and byte locking, the data, as a 42 bit word, is converted to an eight bit word (one byte). Before the 42 bit word is converted, it is multiplied by a quantization gain which toggles the bits to an acceptable point for conversion. We found that increasing the quantization gain resulted in a more accurate depiction of the $T_{sys}$ versus frequency as seen in Figure 5.2.

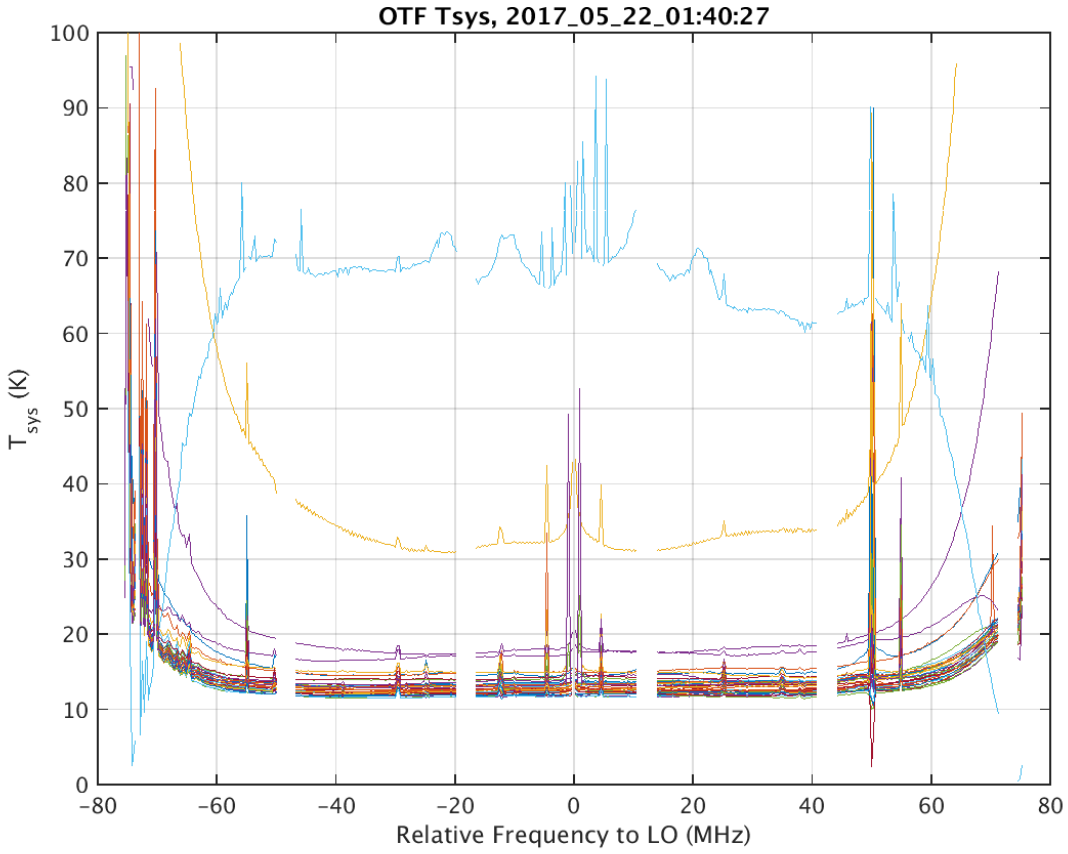



Figure 5.2: $T_{sys}$ (K) vs Relative LO frequency (MHz). Each curve represents the $T_{sys}$ of a single element. The quantization gain was set to 20 which shows an increase in the transition band.

The $T_{sys}$ increases at the edges of the band as expected due to the increase in quantization gain to 20. The irregular curves are due to faulty elements of the PAF. One of the elements was dead and the other two were not well connected. Figure 5.3 shows an increased quantization gain of 40.

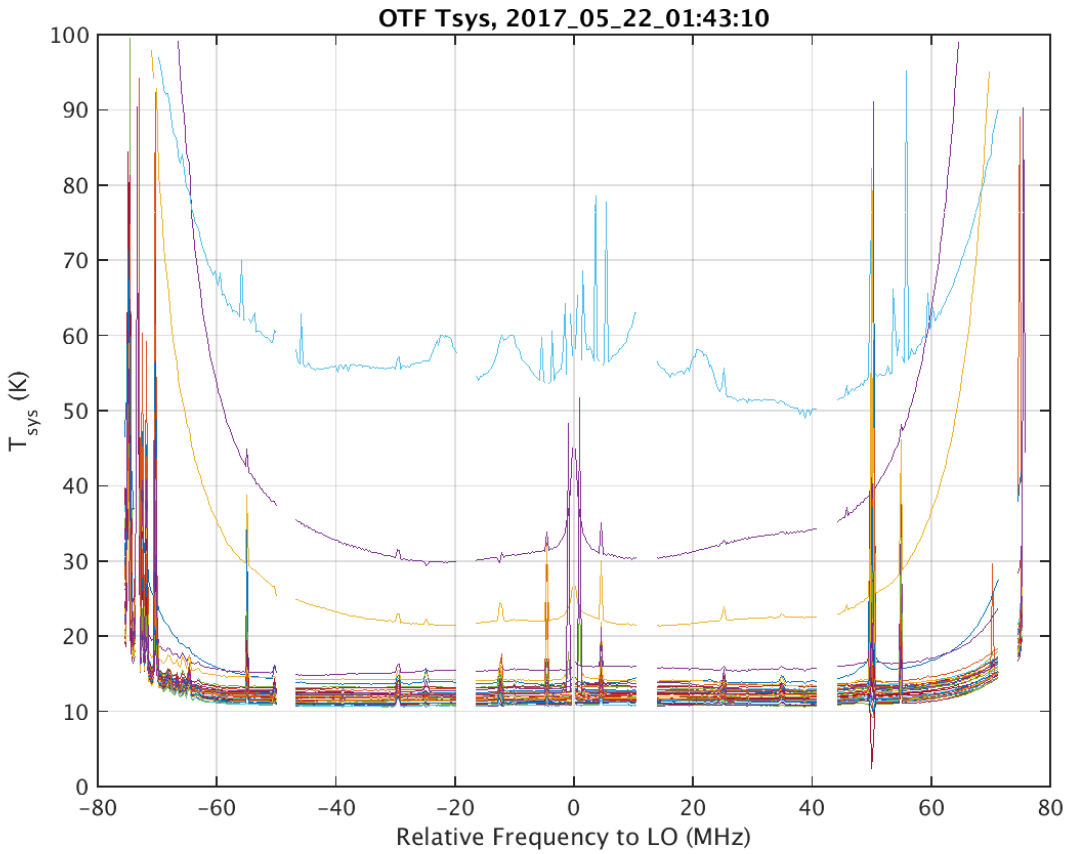



Figure 5.3: $T_{sys}$ (K) vs Relative LO frequency (MHz). Each curve represents the $T_{sys}$ of a single element. The quantization gain was set to 40 and produced lower $T_{sys}$ results in the pass band relative to Figure 5.2.

A quantization gain of 40 produced accurate $T_{sys}$ results and was the gain chosen when acquiring data with the GBT.

## 5.4 GBT Results

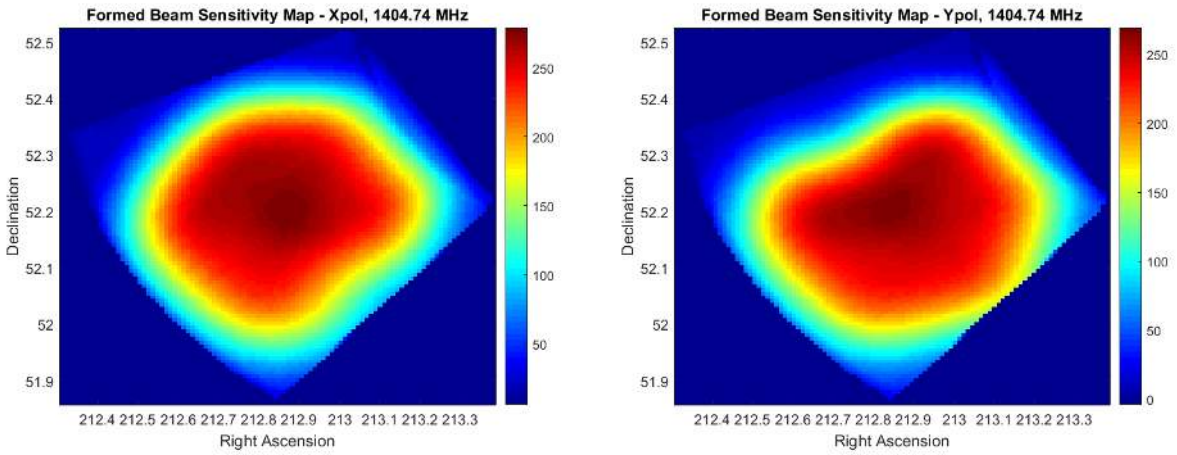
When the PAF was placed on the feed of the GBT, we were given five sessions to acquire data and three of them will be discussed (session two, three, and five). The X polarization elements from session one malfunctioned so the data obtained was unreliable. The results from session five were similar to session four so only the latter of the two are discussed. During each session, we collected data from multiple scans discussed before hand. Each session contained peak and focus scans used to determine pointing and focus offsets respectively, calibration scans used to calculate beamformer weights, and other scans of interest such as; pulsar and HI scans.

After each session, post processing codes were run in order to analyze the data. Data was analyzed using sensitivity map, system temperature, beam pattern, and element pattern plots that will be shown after a description of the session.

During the second session, we were able to perform a continuous $30 \times 30$ PAF calibration scan, and attempted a pulsar scan. A continuous $30 \times 30$ PAF calibration scan is one with $30 \times 30$ on pointings on a grid. For this scan we decided to have an off pointing every other row. The on and off pointings, in this case, are points on and off the grid respectively, formed from the scan. Each pointing represents a beam and each of them is three arcminutes (one arcminute is $1/60$ of a degree) wide. The data from this calibration scan was acquired using the coarse channel correlator and the source used for calibration was 3C295.

Unfortunately, the pointing offsets were inaccurate with the off pointings at a lower elevation than the grid. And the covariances in post processing were calculated using the closest off-pointing to a corresponding row so they were calculated using only one which would not be the case with accurate pointing offsets. This issue was not resolved during this commissioning.

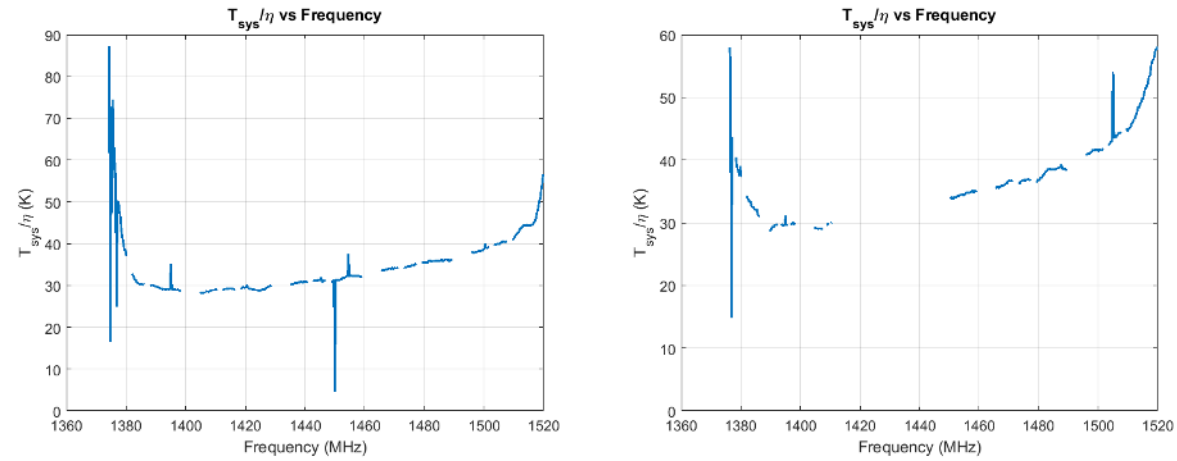
Additionally, the coordinates provided by the telescope coordinator were in right ascension (RA) and declination (DEC). Therefore the sensitivity map does not look as it would with elevation and cross-elevation offsets that can be seen in the next sessions. While the pointing offsets were inaccurate and the coordinates were in RA and DEC, this was the session with the lowest recorded $T_{\text{sys}}/\eta$ of 28.2 K. The sensitivity map and $T_{\text{sys}}/\eta$ plot formed from the calibration grid can be seen in Figure 5.4 and 5.5 respectively.




(a) X polarization

(b) Y polarization

Figure 5.4: Formed beam sensitivity maps of X and Y polarizations at 1404.74 MHz from session 2. This is a $30 \times 30$ grid with beams that were each three arcminutes wide. The grid orientation is due to the RA and DEC coordinate system.



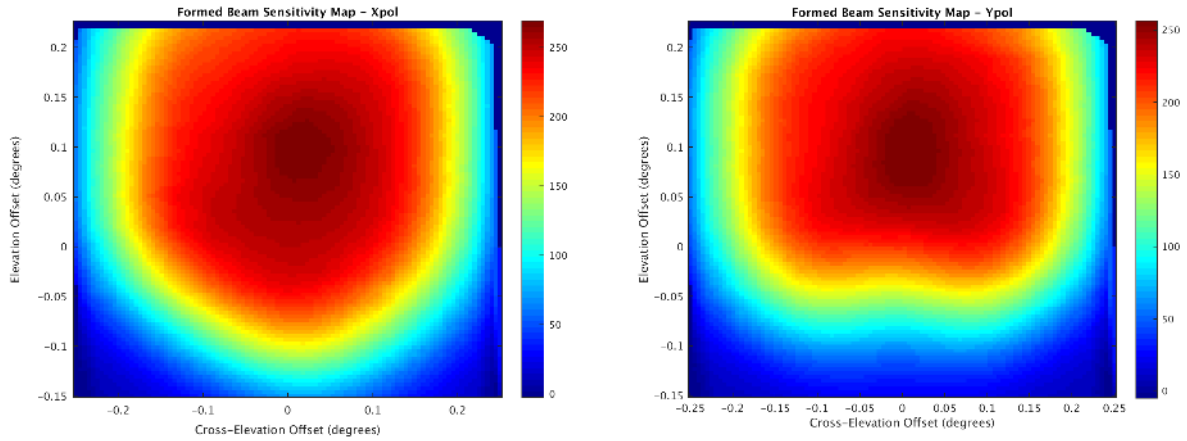

(a) X polarization

(b) Y polarization

Figure 5.5: $T_{\text{sys}}/\eta$ of X and Y polarizations from session 2. The lowest $T_{\text{sys}}/\eta$ was 28.2 K in the X polarization. The blank spots were the missing bandwidth from the unused HPCs as well as masking of RFI.

During the third session, we performed the same continuous $30 \times 30$ calibration scan along with an attempt at an HI source scan. Unfortunately, the fine channel correlator had a few issues to

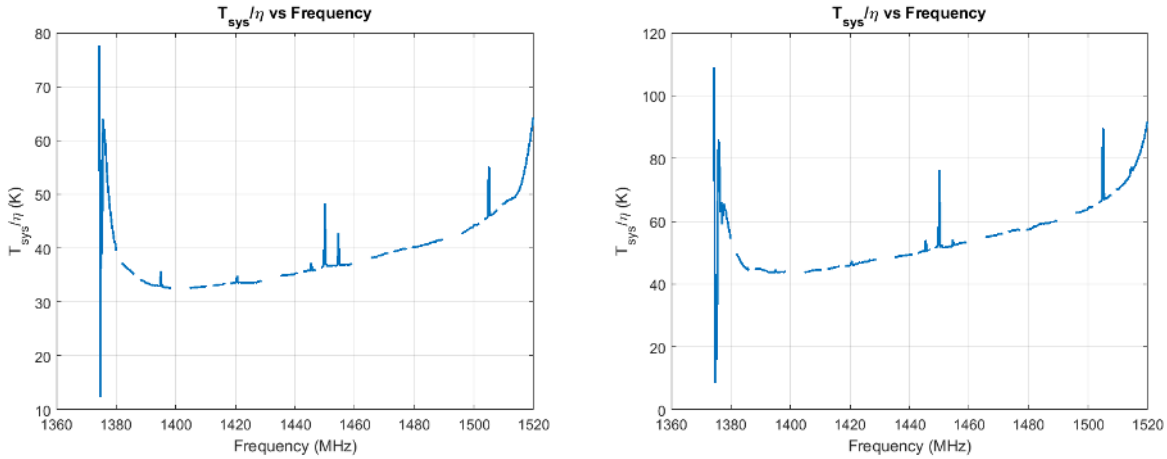
resolve so the most useful data acquired from this session was that of the coarse channel correlator. This grid was generated with elevation and cross-elevation offsets rather than RA and DEC so it looks as it should. The grid along with the $T_{sys}/\eta$ plot of both the X and Y polarizations can be seen in the figures below.




(a) X polarization

(b) Y polarization

Figure 5.6: Formed beam sensitivity maps of X and Y polarizations at 1404.74 MHz from session 3. This is a $30 \times 30$ grid with beams that were each three arcminutes wide.

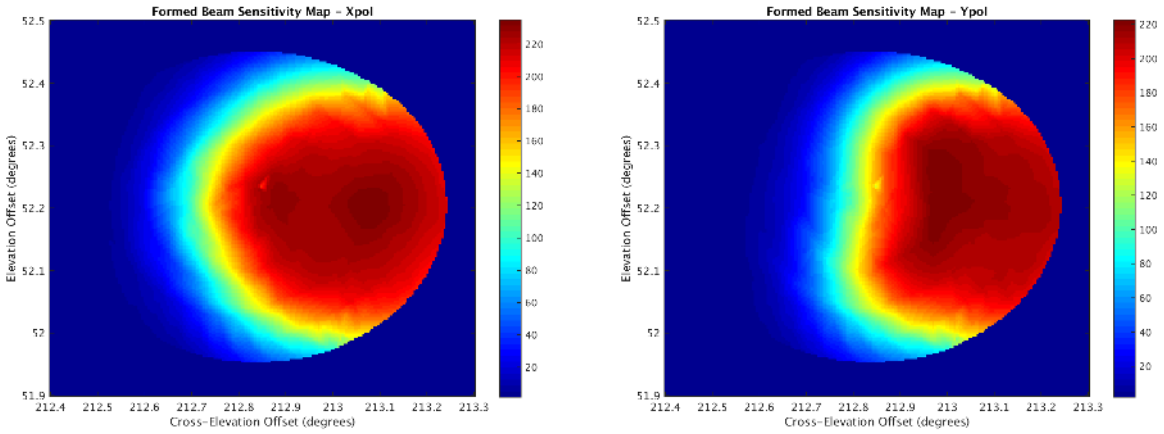



(a) X polarization

(b) Y polarization

Figure 5.7: $T_{\text{sys}}/\eta$ of X and Y polarizations from session 3. The lowest $T_{\text{sys}}/\eta$ was 32.6 K in the X polarization. The blank spots were the missing bandwidth from the unused HPCs as well as masking of RFI.

In the forth session, we acquired a grid with a daisy scan which is performed by drifting the telescope along a petal-like structure. We also acquired seven discrete pointing scans around a calibrator source (3C295) in order to produce quick weight files to run the real-time beamformer and track pulsar sources. The daisy scan was tested because it is a faster scan than the continuous calibration scan (also called a raster scan), but the amount of data stored was large enough to greatly slow down post processing. The daisy scan was not used in the next commissioning because of this problem. The figures below show the sensitivity map using the daisy as well as the $T_{\text{sys}}/\eta$.




(a) X polarization

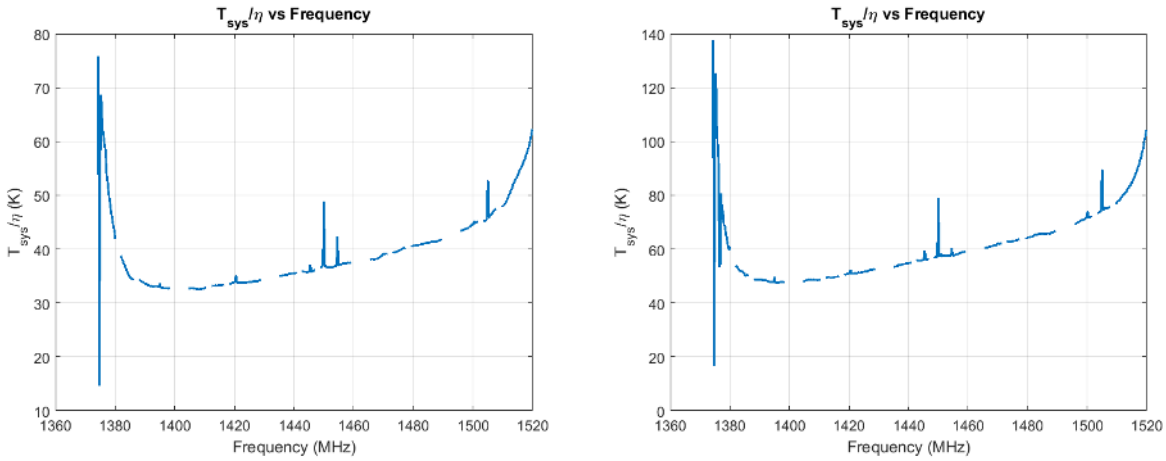
(b) Y polarization

Figure 5.8: Formed beam sensitivity maps of X and Y polarizations at 1404.74 MHz from session 4. These were the daisy scans generated by drifting the telescope along the petal-like structure. This is a $30 \times 30$ grid with beams that were each three arcminutes wide.

Using the steering vectors and weights acquired from the sensitivity/SNR calculation, we were able to plot beam patterns to characterize the beams, and plot element patterns as well. Both patterns were derived with the same equation, but different weights as seen in equation 4.2.

$$p_{j,k,l} = |\mathbf{w}_{j,k,l}^{H} \hat{\mathbf{a}}_{k,l}(\theta)|^2, \tag{5.2}$$

where $j$ is the element index and only changes when generating element patterns, $k$ is the beam index, and $l$ is the frequency index.




(a) X polarization

(b) Y polarization

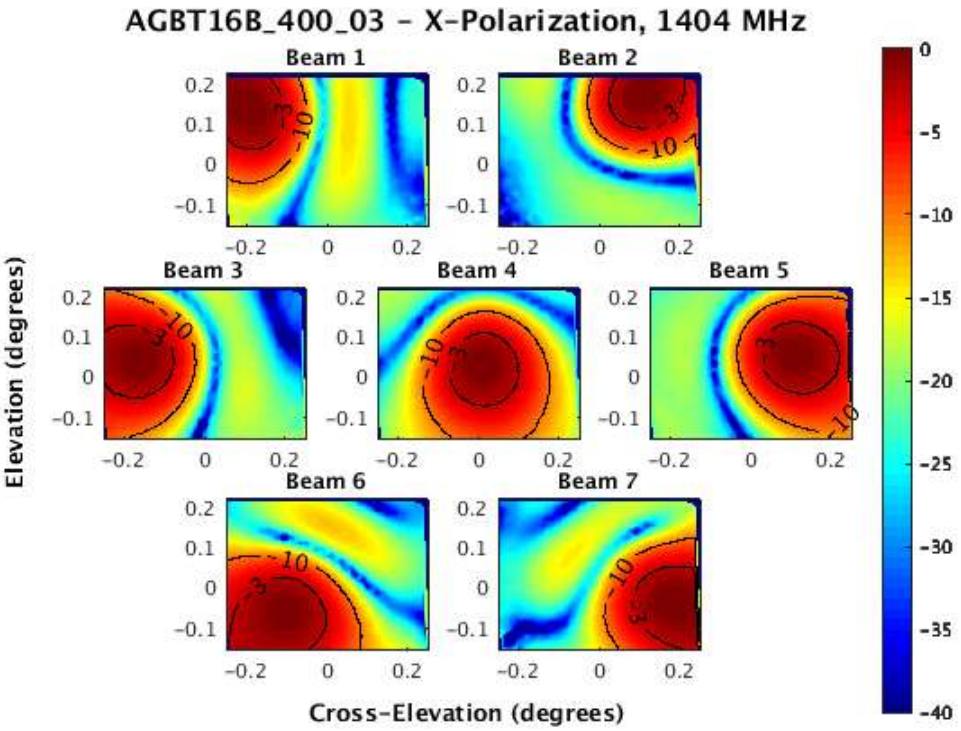
Figure 5.9: $T_{\text{sys}}/\eta$ of X and Y polarizations from session 4. The lowest $T_{\text{sys}}/\eta$ was 32.5 K in the X polarization. The blank spots were the missing bandwidth from the unused HPCs as well as masking of RFI.

The weights used to generate beam patterns were calculated using equation 2.8 while those used to generate element patterns were set to zero in every element except the one being analyzed which was set to one. The beam and element patterns of this commissioning can be seen in Figure 5.10 and 5.11 respectively.

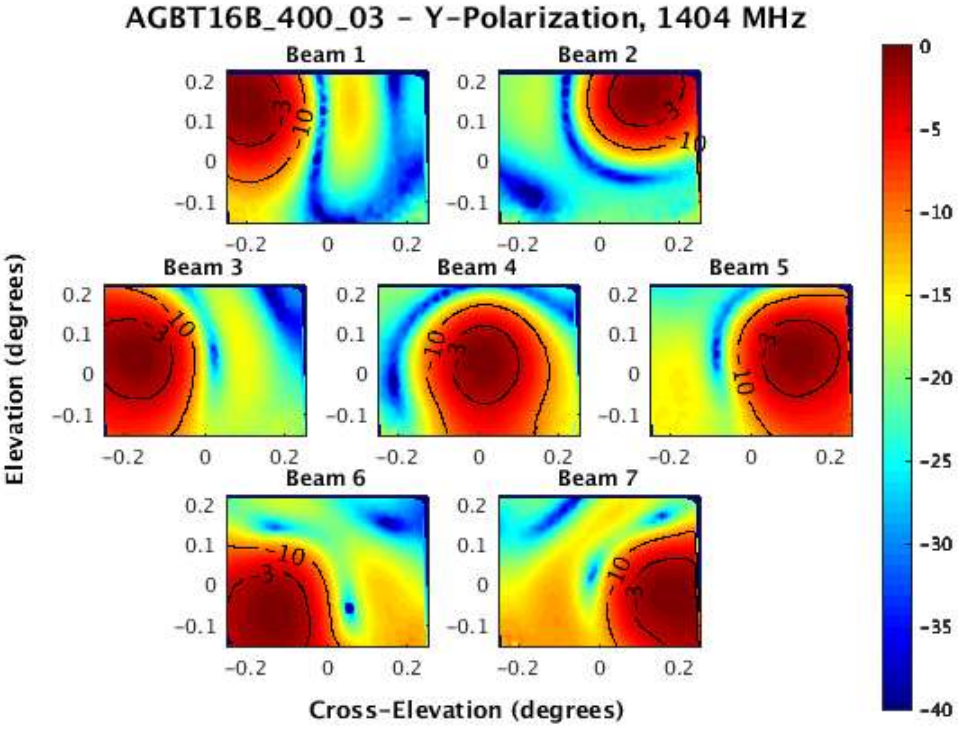
Using the weights acquired from the seven discrete pointing scans, we were also able to track two pulsars, B1933 and B1937, for 10 minutes each. Figure 5.12 shows plots of B1937 which were generated by Kaustubh Rajwade, a PhD student from West Virginia University. The signal is seen to move across frequency as the time samples advance (before dedispersion which aligns the signal across time samples and sums across frequency channels) [7], [30]. This is expected of a pulsar.

## 5.5 Summary

During the May 2017 commissioning, although we had issues with the coordinate system and inaccurate pointing offsets, these tests went quite well. We were able to track pulsars, B1933 and B1937, with the real-time beamformer, and further test the coarse channel correlator.

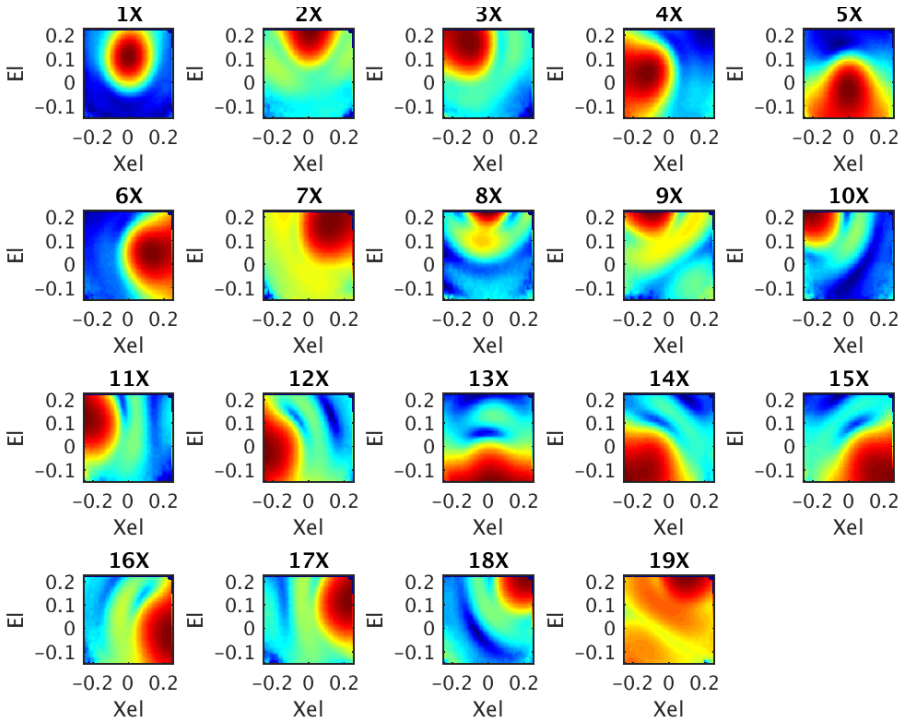



(a) X polarization

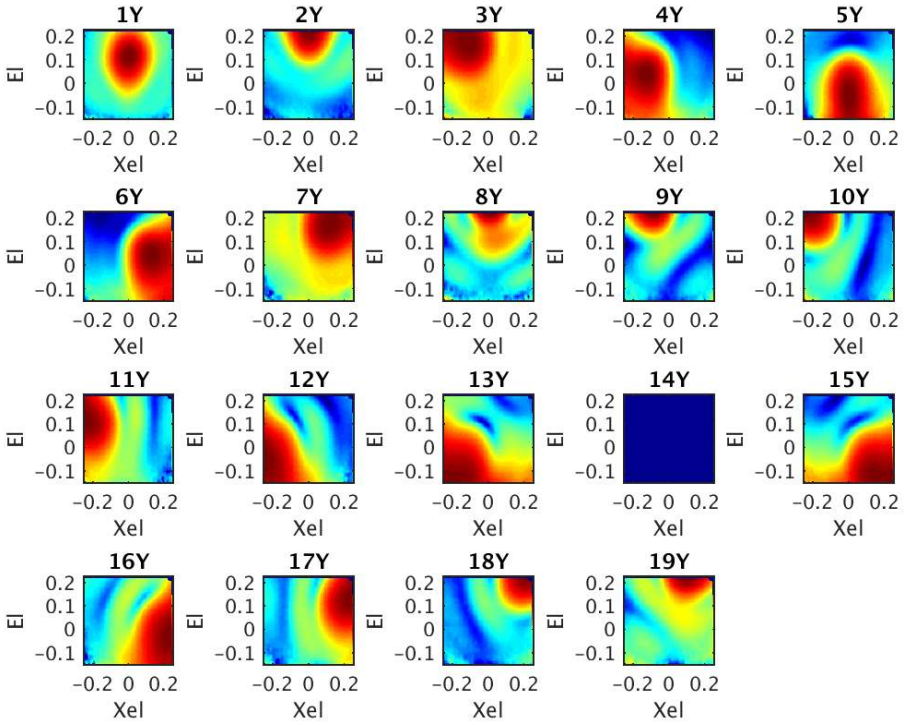



(b) Y polarization

Figure 5.10: Beamformed patterns of X and Y polarizations at 1404.74 MHz for session 3.




(a) X polarization




(b) Y polarization

Figure 5.11: Element patterns of X and Y polarizations at 1404.74 MHz for session 3. The blank element pattern is due to a non-responsive element.

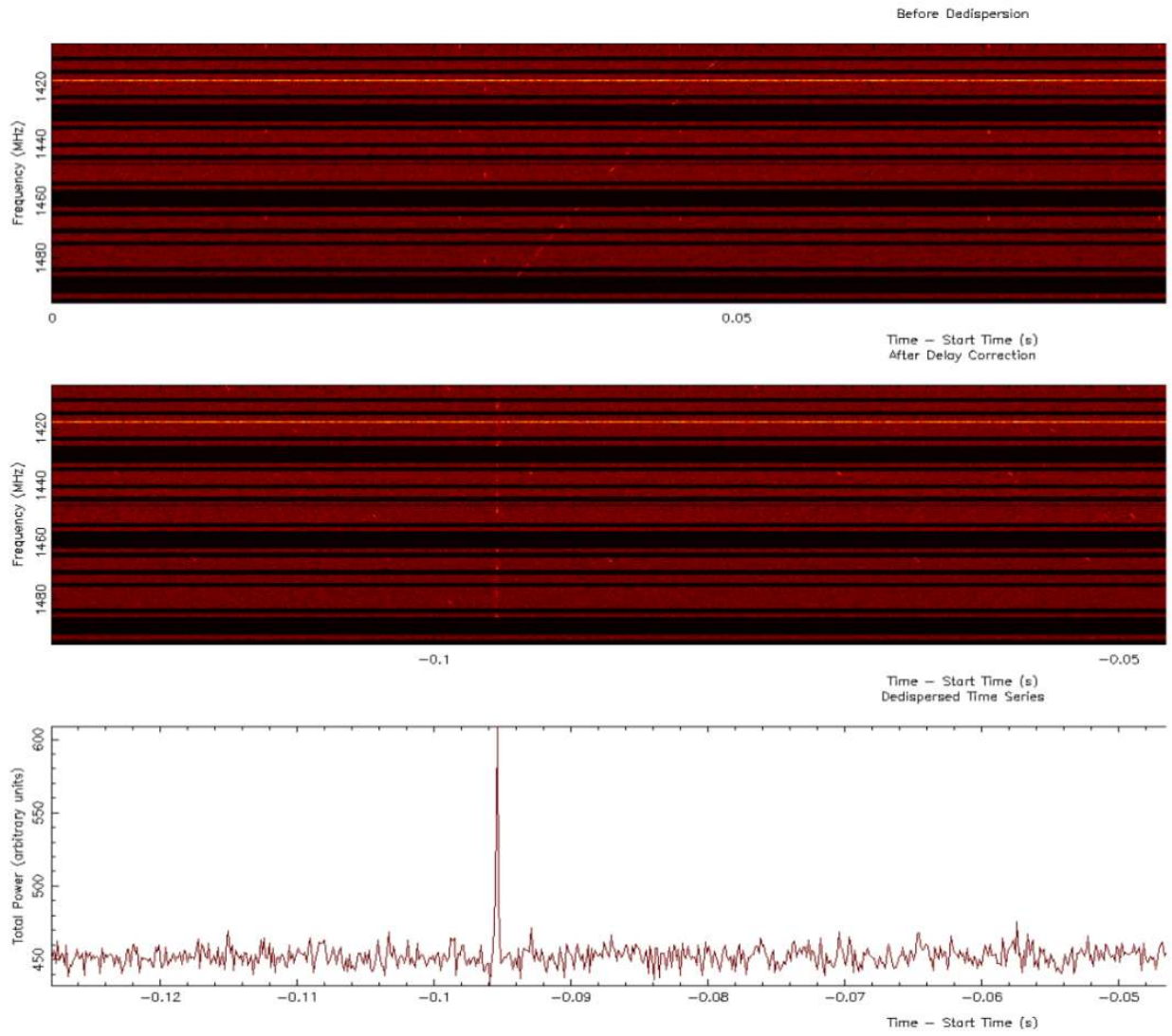



Figure 5.12: Pulsar (PSR B1937+21) detection in beam 1. Before dedispersion (top), dedispersed (middle), and power integrated over frequency (bottom).

The tests run in the OTF proved that we needed to use a quantization gain of 40 for coarse channel correlation. The quantization gain still needed to be tested with the fine channel correlator and in the next commissioning, it was found to be 10.

We were also able to achieve a $T_{\rm sys}/\eta$ of 28.2 K during one of our sessions. This showed that the PAF was capable of achieving system temperatures of a single pixel feed (the $T_{\rm sys}/\eta$ goal was 28 K). Finally, the post-processing code developed during this commissioning enabled us to use our time more efficiently in July 2017.

# CHAPTER 6. JULY 2017 FLAG COMMISSIONING

## 6.1 Introduction

In this chapter, the work done, and results acquired during the July 2017 commissioning of FLAG are discussed. Details on the improvements made to the system as well as those that still need to be made are addressed. As in the previous two chapters, details on the data acquired during this commissioning are discussed as well. In terms of system performance, this appeared to be the best of the three commissioning runs.

While there were still a few unresolved issues, this commissioning proved to be the best of the three. A lot of the improvement was due to the changes made during and after the May 2017 commissioning. Changes made to the HASHPIPE code managed to improve data acquisition issues that were problematic during the last commissioning. Code was also written to correct sample delay from element to element on the array. This was called word lock and is described in detail in [26].

The availability of the post-processing code was one of the most important changes/additions that made the run quite smooth. Pointing offset issues were also resolved which lead to accurate cross-elevation and elevation values from the calibration grid to be used for post-processing. Unfortunately, an issue inherent to the PFB was discovered prior to this commissioning and due to time constraints was not solved, but solutions have been addressed.

The data acquisition procedure remained the same as the last commissioning except word lock needed to be done after bit/byte lock. Bit/byte lock was still done manually using the same GUI from the May 2017 commissioning.

## 6.2 DIBAS Modifications

Quite a few modifications were made to the DIBAS code to make it more efficient as well as fix some issues seen in previous commissioning runs. One issue that has been consistent through all commissioning runs was problematic data transfer between HASHPIPE threads.

Between each HASHPIPE thread is a semaphore controlled ring buffer which transfers data to the next thread based on whether the buffer is full or empty (clean). The problem is that during data acquisition, the threads did not seem to keep up with each other. So we received some blocks of data with the incorrect number of packets (which we refer to as bad blocks) or data acquisition stops before a scan is complete (which we refer to as stalling). And sometimes a stream of bad blocks would cause stalling.

The major reason for the large number of bad blocks and stalling was the FITS writer. The problem was that for all instances of HASHPIPE on an HPC, the FITS writers were using the same processor cores. This caused a higher demand of memory access that the HPCs could not handle. After distributing the FITS writers to different cores on the HPC, there was a significant decrease in bad blocks as well as stalling. Therefore, we were able to acquire data over a higher bandwidth.

Despite an increase in bandwidth with the FITS writer fix, there was still stalling and a few bad blocks mainly in the real-time beamformer due to its high data rate. To increase the bandwidth further, HASHPIPE and CUDA code had to be optimized.

There were a few memory leaks (memory that was not freed) from the initialization in the code. This was easily fixed by freeing the memory after each run of the beamformer. The major change was integrating the algorithm used in the transpose thread into the data restructure kernel in the CUDA code.

While debugging, the transpose thread was seen to run much longer than the other threads. This seemed to be causing much of the stalling while running the beamformer. So the transpose thread was eliminated, and its algorithm was integrated into the data restructure kernel of the beamformer. The algorithm used in the transpose thread restructured the data to combine all 40 input ports from the five ROACHES. The equations below show output of the net thread (input of the transpose thread) and the output of the transpose thread.

```
Net = i + Ni*c + Nc*Ni*t + Nt*Nc*Ni*f + Nf*Nt*Nc*Ni*m,
```

$$Tr = e + Ne*c + Nc*Ne*t + Nt*Nc*Ne*m,$$

where Ni is the number of inputs per F-engine/ROACH and i are the input indices, Nc is the number of frequency channels and c are the channel indices, Nt are the number of time samples and t are the sample indices, Nf are the number of F-engines/ROACHES and f are the F-engine indices, and Ne are the total number of elements/input ports and e are the element indices.

Integrating the transpose thread into the CUDA code reduced the number of stalled instances to the point that we lost at most 20% of the bandwidth rather than 50%. Although this was a great improvement in performance, the stalling issue still needs to be completely resolved. Two of the possible solutions that are still being tested and understood are changing the buffer sizes and HASHPIPE thread optimization.

### 6.3 Pointing Offset Correction

Some modifications were made to the post processing code, specifically when calculating the cross-elevation and elevation offsets. Initially, the cross-elevation was inaccurately calculated as though it was azimuth. In order to calculate these pointing offsets, encoder values (antenna position) as well as the observer's command positions (on-sky beam position) were used.

The encoder and observer command elevation/cross-elevation offsets were calculated first. This is done to estimate the actual elevation and cross-elevation offsets. The encoder and observer elevation offsets were calculated as follows

$$mntEloff = mntEl - smntcEl,$$
$$obscEloff = obscEl - sobscEl,$$

where mntEl and smntcEl are the encoder elevation at the point in a scan and command at the scan midpoint respectively. obscEl and sobscEl are the observer elevation at the point in a scan and command at the scan midpoint respectively.

The encoder and observer cross-elevation offsets were calculated as follows

$$mntXEloff = (mntAz - smntcAz)cos(mntEl),$$
$$obscXEloff = (obscAz - sobscAz)cos(mntEl),$$

where `mntAz` and `smntcAz` are the encoder azimuth at the point in a scan and command at the scan midpoint respectively. `obscAz` and `sobscAz` are the observer azimuth at the point in a scan and command at the scan midpoint respectively. Since we are calculating the cross-elevation rather than azimuth, the difference is multiplied by the cosine of the encoder elevation.

The elevation and cross-elevation offsets were calculated by taking the difference between the encoder and observer values as follows

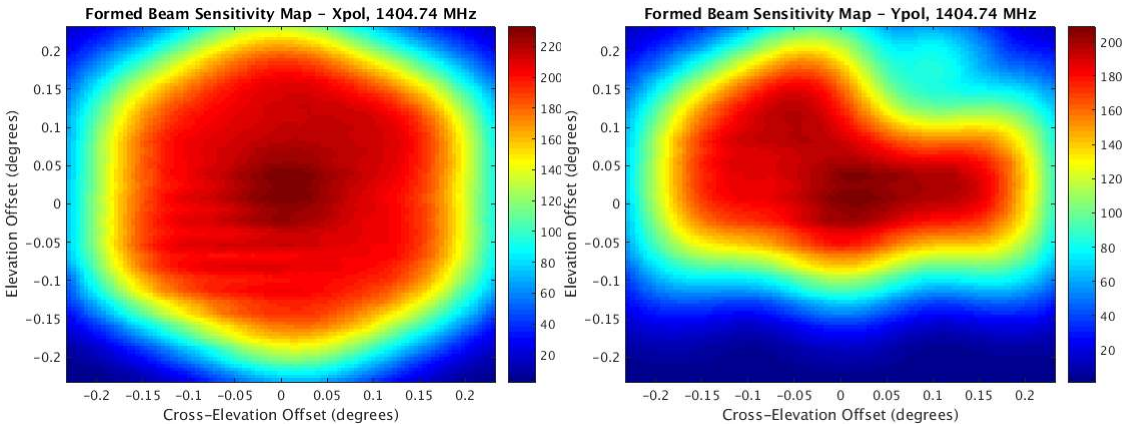
```
elOff = mntEloff - obscEloff,
xelOff = mntXEloff - obscXEloff.
```

These modifications, as well as a few others, ensured that we were able to process the data more efficiently and accurately than the last commissioning. However, the results, while good, were not as impressive as they were in the last commissioning. The sensitivity during this commissioning was slightly higher than the last commissioning overall.

### 6.4 GBT Results

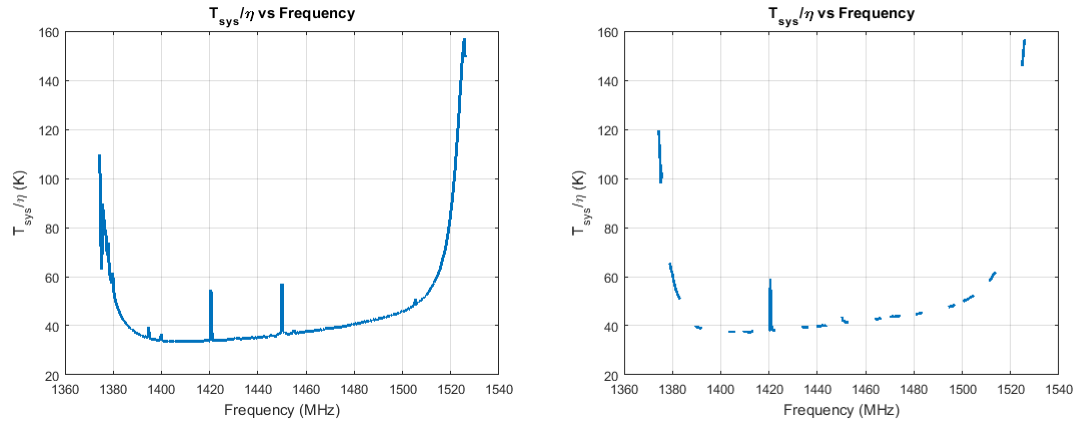
During this commissioning, we were able acquire and process data from three sessions. The procedure for data acquisition remained the same as previous commissioning runs, except for the additional word lock step. Sources 3C295, 3C123, and 3C147 were used for calibration during each session respectively. Unfortunately, the data acquired with the real-time beamformer during this run appeared to be inaccurate. There seemed to be significantly high power at the edges of the frequency band compared to the center. This issue is still under investigation with the problem possibly being low quantization gain.

While three different sources were used, one session's plots can be used to illustrate the results from the run. A continuous $30 \times 30$ calibration scan was used during each session. An off pointing was placed every five rows rather than every other row as in the previous run. Due to the pointing offset corrections applied during this run, the plots used for analysis were better aligned in elevation and cross-elevation compared to the previous run. The sensitivity map and $T_{\text{sys}}/\eta$ for the second session can be seen in Figure 6.1 and 6.2 respectively.




(a) X polarization (b) Y polarization

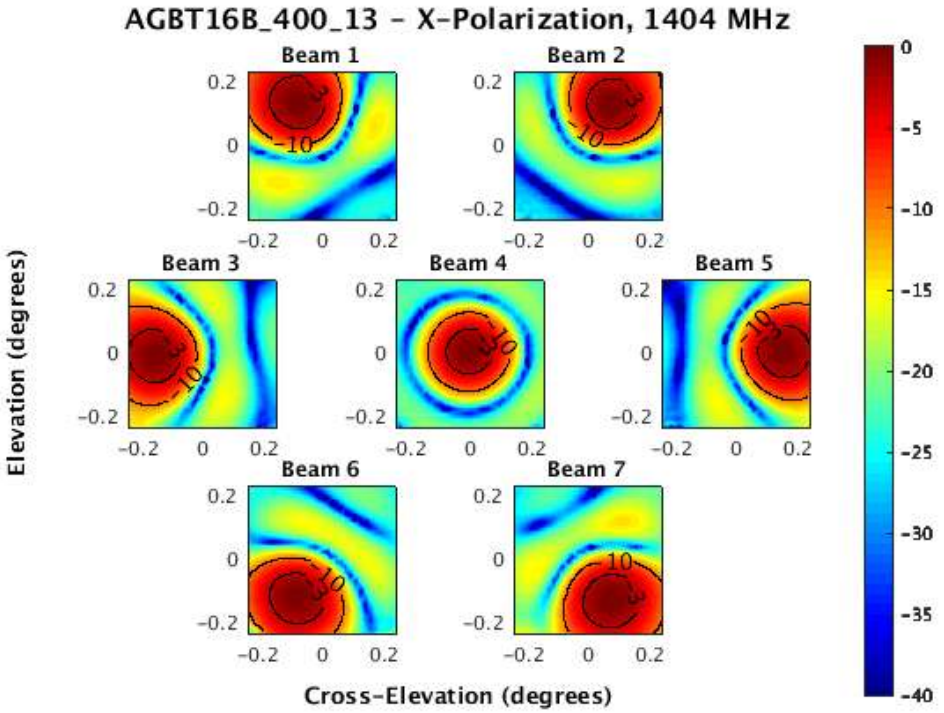
Figure 6.1: Formed beam sensitivity maps of X and Y polarizations at 1404.74 MHz from session 3. This is a $30 \times 30$ grid with beams that were each three arcminutes wide.



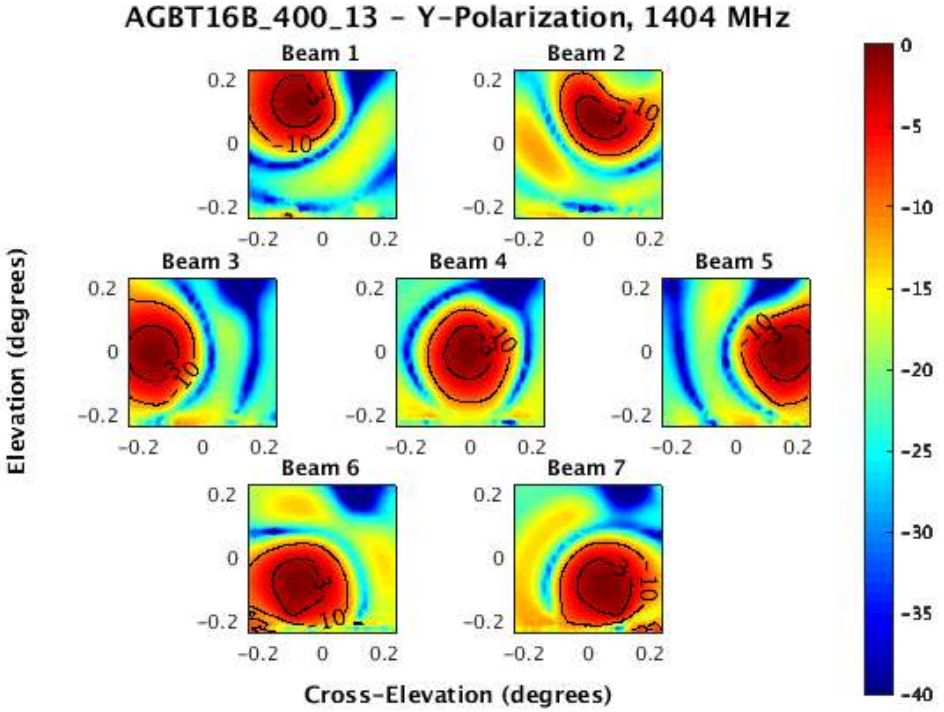

(a) X polarization (b) Y polarization

Figure 6.2: $T_{\text{sys}}/\eta$ of X and Y polarizations from session 3. The blank spots were the missing bandwidth from stalled HASHPIPE instances as well as masking of RFI.

As with the previous sessions, quite a few of the Y-polarized elements were non-responsive. The sensitivity map with the Y-polarized elements is indicative of this issue. This can also be seen in the beam and element patterns in Figure 6.3 and 6.4 respectively.

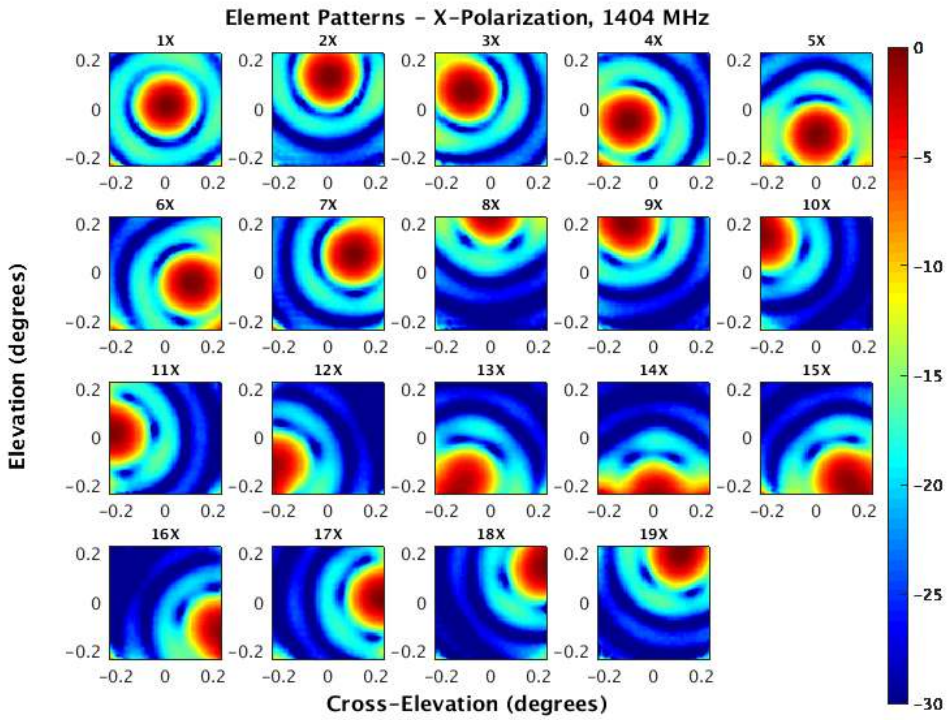



(a) X polarization

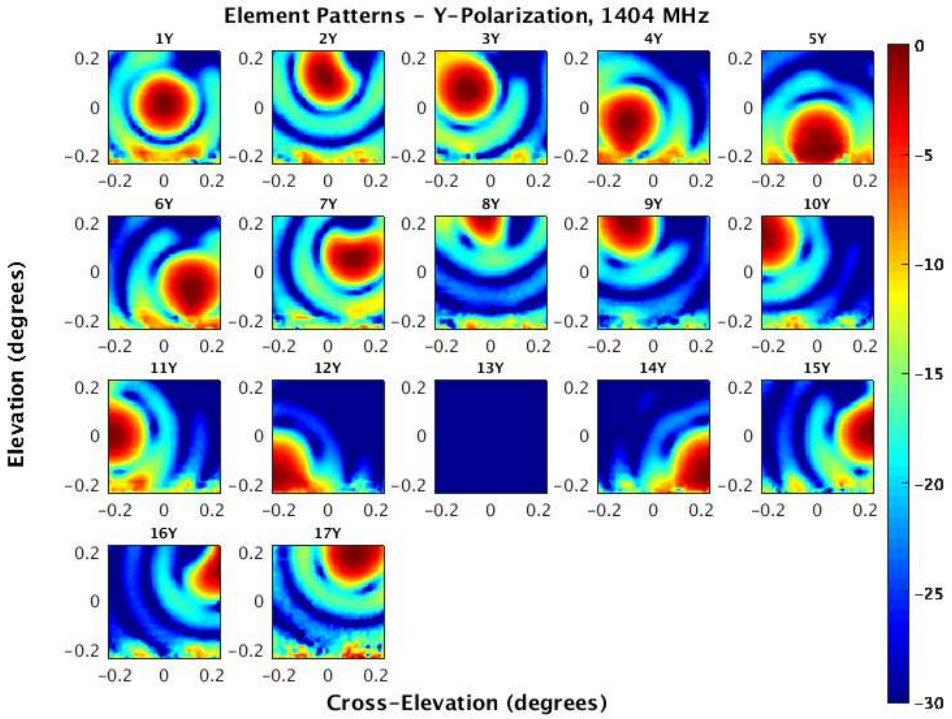



(b) Y polarization

Figure 6.3: Beamformed patterns of X and Y polarizations at 1404.74 MHz for session 3.




(a) X polarization




(b) Y polarization

Figure 6.4: Element patterns of X and Y polarizations at 1404.74 MHz for session 3.

## 6.5 Summary

The July 2017 commissioning was more efficient than the previous two due to the additions and improvements made to the back end between commissioning runs. Despite the issue with the real-time beamformer output, we were able to acquire data in all modes with little problem. We were also able to generate an HI map which was not done in the previous commissioning [26].

After the problem with the beamformer output is solved, more improvements will be made to the system. A GUI has been created, but has not been fully tested. And the HASHPIPE code needs to be modified to increase speed and improve data integrity. This can be done by eliminating the need to copy data to and from the GPU and modifying the HASHPIPE threads.

# CHAPTER 7. CONCLUSION AND FUTURE WORK

In radio astronomy, phased array feeds are used to achieve larger fields of view than the traditional single pixel feed. This has been achieved with FLAG, and as expected with a PAF, greater complexity in the receiver and higher data rate are introduced.

The data is transmitted from the front end receiver to the digital back end through optical fibers at a data rate of 303.75 kHz. The data is then processed and packetized by ROACH II boards, and a 10 Gbe switch respectively. The data from the switch is then processed by five HPCs each containing two GPUs. Each GPU processes $1/10$ of the receiver bandwidth. The code implemented on the GPUs is integrated into a real-time operating system called HASHPIPE. This RTOS is run by a user interface called dealer/player that runs the appropriate mode for a scan. During a scan, the modes vary from coarse/fine channel correlation, beamforming or commensal mode which is when the correlator and beamformer are run in parallel.

During observations made in July 2016, May 2017, and July 2017, we were able to test the FLAG back end. Data was acquired and verified by analyzing the sensitivity over a field of view and system temperature over the 150 MHz bandwidth. These values are obtained by calculating weights and using the correlation matrices acquired from the correlator. The weights are also used for real-time beamforming when detecting radio transient sources.

In May 2017, a pulsar (PSR B1937+21) was detected with the real-time beamformer proving that FLAG is capable of detecting radio transient sources.

## 7.1 Future Work

A few additions and improvements to the system still need to be made or tested. The specifications of FLAG were to design a back end with the fine/coarse correlator and beamformer working in parallel. This mode is referred to as commensal mode, and it has not been tested yet. This should be one of the priorities moving forward with FLAG.

Bit/byte/word lock also needs to be fully integrated into the code in order to remove the time wasting manual procedure. This has already been done and just needs to be tested.

A weight generation/calibration thread should be added to the system to reduce the time spent post-processing. Currently, weight files are generated in MATLAB using data from the correlator after a calibration scan. A thread should be added to HASHPIPE that quickly calculates weights using the GPUs. Real-time analysis tools should also be incorporated into the system to provide live updates as well as reduce the time spent post-processing even more.

In order to greatly speed up processing, the GPUs need to be better utilized. The CUDA code needs to be optimized in such a way that the number of memory copies to and from the device are greatly reduced. This could possibly be done by integrating HASHPIPE into the GPUs which at the very least would eliminate the initial copy from host to device.

## REFERENCES


[1] M. Elmer, "Improved Methods for Phased Array Feed Beamforming in Single Dish Radio Astronomy," Ph.D. dissertation, Brigham Young University, 2012. 1, 2

[2] J. R. Fisher, "Phased Array Feeds for Low Noise Reflector Antennas," *NRAO Electronics division internal report*, no. 307, 1996. 1

[3] B. D. Jeffs, K. F. Warnick, J. Landon, J. Waldron, D. Jones, J. R. Fisher, and R. D. Norrod, "Signal Processing for Phased Array Feeds in Radio Astronomical Telescopes," *IEEE Journal of Selected Topics in Signal Processing*, vol. 2, no. 5, pp. 635–646, 2008. 1, 4, 5, 7

[4] J. Landon, M. Elmer, J. Waldron, D. Jones, A. Stemmons, B. D. Jeffs, K. F. Warnick, J. R. Fisher, and R. D. Norrod, "Phased Array Feed Calibration, Beamforming, and Imaging," *The Astronomical Journal*, vol. 139, no. 3, p. 1154, 2010. 1, 4, 5, 7

[5] M. Elmer, B. D. Jeffs, K. F. Warnick, J. R. Fisher, and R. D. Norrod, "Beamformer Design Methods for Radio Astronomical Phased Array Feeds," *IEEE Transactions on Antennas and Propagation*, vol. 60, no. 2, pp. 903–914, 2012. 1, 4, 5, 7

[6] B. D. Van Veen and K. M. Buckley, "Beamforming: A Versatile Approach to Spatial Filtering," *IEEE assp magazine*, vol. 5, no. 2, pp. 4–24, 1988. 1, 4, 5, 7

[7] C. Faucher-Giguere and V. M. Kaspi, "Birth and Evolution of Isolated Radio Pulsars," *The Astrophysical Journal*, vol. 643, no. 1, p. 332, 2006. 1, 46

[8] D. Lorimer, M. Bailes, M. McLaughlin, D. Narkevic, and F. Crawford, "A Bright Millisecond Radio Burst of Extragalactic Origin," *Science*, vol. 318, no. 5851, pp. 777–780, 2007. 1

[9] D. e. a. Thornton, B. Stappers, M. Bailes, B. Barsdell, S. Bates, N. Bhat, M. Burgay, S. Burke-Spolaor, D. Champion, P. Coster *et al.*, "A Population of Fast Radio Bursts at Cosmological Distances," *Science*, vol. 341, no. 6141, pp. 53–56, 2013. 1

[10] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," in *Application Specific Processors, 2008. SASP 2008. Symposium on*. IEEE, 2008, pp. 101–107. 1

[11] D. Lorimer, M. McLaughlin, D. J. Pisano, R. Prestage, and A. Roshi, "FLAG Beamformer Specifications," 2014. 1, 8

[12] R. A. Black, "Digital Back End Development and Interference Mitigation Methods for Radio Telescopes with Phased Array Feeds," Master's Thesis, Brigham Young University, 2014. 2

[13] A. Parsons, D. Backer, A. Siemion, H. Chen, D. Werthimer, P. Droz, T. Filiba, J. Manley, P. McMahon, A. Parsa *et al.*, "A Scalable Correlator Architecture Based on Modular FPGA

Hardware, Reuseable Gateware, and Data Packetization," *Publications of the Astronomical Society of the Pacific*, vol. 120, no. 873, p. 1207, 2008. 2

[14] A. Parsons, D. Backer, C. Chang, D. Chapman, H. Chen, P. Crescini, C. De Jesus, C. Dick, P. Droz, D. MacMahon *et al.*, "Petaop/Second FPGA Signal Processing for SETI and Radio Astronomy," in *Signals, Systems and Computers, 2006. ACSSC'06. Fortieth Asilomar Conference on*. IEEE, 2006, pp. 2031–2035. 2, 8

[15] V. Asthana, "Development of L-band Down Converter Boards and Real-Time Digital Back End for Phased Array Feeds," Master's Thesis, Brigham Young University, 2012. 2

[16] T. Webb, "Design and Polarimetric Calibration of Dual-Polarized Phased Array Feeds for Radio Astronomy," Master's Thesis, Brigham Young University, 2012. 2

[17] K. Rohlfs and T. Wilson, *Tools of Radio Astronomy*. Springer Science & Business Media, 2013. 4

[18] F. Haddock, "Introduction to Radio Astronomy," *Proceedings of the IRE*, vol. 46, no. 1, pp. 3–12, 1958. 4

[19] J. Diao, "High Sensitivity Phased Array for Radio Astronomy and Satellite Communications," Ph.D. dissertation, Brigham Young University, 2017. 8

[20] M. A. Morgan and J. R. Fisher, "Experiments with Calibrated Digital Sideband-Separating Downconversion," *Publications of the Astronomical Society of the Pacific*, vol. 122, no. 889, p. 326, 2010. 8

[21] "Roach-2 Revision 2," Roach-2 revision 2. https://casper.berkeley.edu/wiki/ROACH-2_ Revision_2, 2013. 8, 30

[22] M. A. Morgan, J. R. Fisher, and J. J. Castro, "Unformatted Digital Fiber-Optic Data Transmission for Radio Astronomy Front Ends," *Publications of the Astronomical Society of the Pacific*, vol. 125, no. 928, p. 695, 2013. 9

[23] D. Macmahon, "HASHPIPE," https://github.com/david-macmahon/hashpipe, 2014. 9

[24] D. C. Wells and E. W. Greisen, "FITS a Flexible Image Transport System," in *Image Processing in Astronomy*, 1979, p. 445. 11

[25] M. Whitehead, "DIBAS Documentation," http://www.gb.nrao.edu/ mwhitehe/dibas/html/introduction.html, 2013. 12

[26] M. C. Burnett, "Advancements in Radio Astronomical Array Processing: Digital Back End Development and Interferometric Array Interference Mitigation," Master's Thesis, Brigham Young University, 2017. 14, 38, 50, 57

[27] C. Nvidia, "NVIDIA CUDA C Programming Guide," *NVIDIA Corporation*, vol. 120, no. 18, p. 8, 2011. 16, 17

[28] ——, "Cublas Library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, no. 27, p. 31, 2008. 18

[29] M. Morgan and J. R. Fisher, “Word Boundary Detection in a Serialized, Gaussian Distributed, White Noise Data Stream,” Electronics Division Technical Note, Tech. Rep., 2009. 31

[30] R. A. Hulse, “The Discovery of the Binary Pulsar,” *Reviews of Modern Physics*, vol. 66, no. 3, p. 699, 1994. 46

# APPENDIX A.

## A.1 Introduction

The following appendix provides details on improvements and additions made to the FLAG back end. The details in this appendix are ones that I have worked on. Others can be found in Richard Allen Black's dissertation, and Mitchell Burnett's thesis.

This appendix will provide details on the hardware, post-processing code, dealer/player interface, and real-time beamformer code.

## A.2 Hardware

### A.2.1 40 Gbe Switch

This is a Mellanox SX 1012 12-port 40 Gbe network switch which redirects packets into the five HPCs so that each HPC receives 100 frequency bins for all 40 elements.

### A.2.2 HPCs

Each HPC is a Mercury GPU408 4U GPU Server purchased from Advanced HPC (part number AH-GPU408-SB14). The machine comes with two Intel Xeon six-core processors organized onto two independent PCI-E busses. A table of specifications for each HPC can be found in Table 1.

Table A.1: HPC Specifications

| **Feature** | **Description** |
|---|---|
| Processors | 2x Intel Xeon E5-2630 v2 2.60 GHz six-core 80 W processors |
| Memory | 32 GB DDR3 ECC |
| Hard Drives | 2x 500 GB SATA 7200 RPM disks in RAID 1 configuration |
| Drive Bays | 8x hot-swappable 3.5 inch drive bays (unpopulated) |
| PCI-E Slots | 4x PCI-E 3.0 x16 slots (double width, two per processor) |
| | 2x PCI-E 3.0 x8 slots (one per processor) |
| | 1x PCI-E 2.0 x4 slot |
| Network | Integrated Intel i350 dual port GbE LAN |
| Power Supply | 1620 W platinum level efficiency redundant power supply |
| Height | 4U |
| Rack Mountable | Yes |

### Graphics Cards

Two models of graphics cards are used in the FLAG back end: the NVIDIA GeForce GTX 780 Ti and NVIDIA GeForce GTX 980 Ti. Two of the GPUs are the 780 Ti model since they were purchased just prior to their abrupt removal from the market by NVIDIA. Pertinent specifications for both cards are summarized in Table A.2 and A.3.

Table A.2: GeForce GTX 780 Ti Specifications

| **Specification** | **Value** |
|---|---|
| # CUDA Cores | 2880 |
| Base Clock Speed | 875 MHz |
| Texture Fill Rate | 210 GigaTexels/sec |
| Memory Clock Speed | 7.0 Gbps |
| Memory | 3 GB |
| Memory Interface Width | 384 bits |
| Memory Bandwidth | 336 GB/sec |
| Minimum Power Requirement | 600 W |

Table A.3: GeForce GTX 980 Ti Specifications

| **Specification** | **Value** |
|---|---|
| # CUDA Cores | 2816 |
| Base Clock Speed | 1000 MHz |
| Texture Fill Rate | 176 GigaTexels/sec |
| Memory Clock Speed | 7.0 Gbps |
| Memory | 6 GB |
| Memory Interface Width | 384 bits |
| Memory Bandwidth | 336.5 GB/sec |
| Minimum Power Requirement | 600 W |

The differences between the two cards in all critical performance specifications are minimal, thus we do not anticipate any new throughput bottlenecks resulting from the swap.

### Network Cards

Each HPC is equipped with three network cards; two dual-port SFP+ 10-GbE cards for receiving packets from the ROACH boards and an Infiniband Card for fully processed data transfers to a Lustre disk array.

## A.3 Post-Processing Code

The post-processing code can be found on any of the Green Bank observatory machines under /home/groups/flag/matFlag/. The following items are a list and description of the files and functions used in post-processing.

sensitivity_map.m This is a matlab file that uses the covariance matrices obtained from a scan using the correlator operational mode, and calculates and plots the sensitivity over the field of view and system temperature. It also generates weights, and steering vectors.

extract_covariances.m This is a function that extracts covariances from the FITS files generated by HASHPIPE when using the correlator operational mode.

get_antenna_positions.m A function used to calculate the elevation and cross-elevation offsets. These help determine the position of the beams on the grid.

get_steering_vectors.m A function used to generate steering vectors.

plot_beam_patterns.m A file that plots 7 beam patterns using the weights generated by sensitivity_map.m. It also creates weight file from the weights.

get_grid_weights.m A function used in plot_beam_patterns.m that selects 7 beams from the weights generated by sensitivity_map.m. The total number of beams correspond to the total number of on-pointings on the grid.

get_element_patterns.m A file that plots the element patterns using the steering vectors generated by sensitivity_map.m

RTBF_data_analysis.m A file that plots output of the real-time beamformer. Provides intensity plots for short time integration (STI) windows vs. frequency bins. It also plots the power over the STI windows.

bf_tf_power.m Also plots output of the real-time beamformer operational mode. Plots power over the frequency bins

extract_bf_output.m Extracts the data from the FITS files generated by the real-time beamformer. These files were too large to be read by the fitsread() function in MATLAB. Low-level functions were used to interact directly with CFITSIO library functions. These functions allow the user to read a fraction of the data from the fits files rather than the entire file.

snoop_lock.m & compare_lock.m Compute the sample delay relative to the first element.

scan_table.m Contains global parameters used by various files.

### A.4 Dealer/Player

This section describes how to run dealer/player with the real-time beamformer operational mode. This procedure is very similar to other modes aside from a few parameter changes.

After a calibration scan, weight files are generated with the post-processing codes. These weight files are used by the real-time beamformer mode.

To run the beamformer:

- Go to the python directory on the HPCs.
- Start the players on the appropriate HPCs. The following line is to be entered in each linux terminal (20 terminals, 4 on each HPC):
  - `ipython player.py "bank name in capital letters e.g.  BANKA"`
- Open another terminal for the dealer (this can be on any HPC) open ipython and in the python directory and enter the following lines:

- `import dealer`
- `d = dealer.Dealer()`
- `d.add_active_players(d.list_active_players())`
- `d.set_mode('FLAG_RTBF_MODE')`
- `d.set_param(weight_file = 'weight file name (Bank added automatically)')`

- If you are running a scan without the scan coordinator:
  - `d.startin('sleep time', 'scan length')`

- If you are running a scan with the scan coordinator:
  - `sc = scanOverlord()`
  - `sc.addDealer(d)`
  - `sc.goLive()`
  - `sc.start_overlord()`

### A.5 Real-time Beamformer

The real-time beamformer is implemented on a Graphics Processing Unit (GPU) using CUDA (A parallel computing platform and application programming interface (API) created by NVIDIA). The specifications for the beamformer are as follows; 19 dipole elements, 25 coarse channels with 15MHz total bandwidth at a sample rate of 303k samp/s, and 7 dual polarization beams. In order to achieve real-time, the total duration of beamforming, and integration must be at most the number of samples (N) divided by the sample rate. The two processes that utilized the GPU the most were beamforming and integration. A data restructure was also processed by the GPU in order to accommodate for a function used by the beamformer. This section describes the functions used to run the real-time beamformer. The code was structured in such a way that it could be easily integrated into HASHPIPE.

### A.5.1 Function Descriptions and Returns

`init_beamformer()` - This function allocates memory to all the arrays used in the code. It also sets up all the arrays used by the `cublasCgemmBatched()` function.

`update_weights(char * filename)` - As the functions name implies, this function updates the weights, but it also transposes the weights to accommodate for the `cublasCgemmBatched()` function. The file name of the weights file is required as an input.

`data_restructure(signed char * data, cuComlplex * data_restruc)` - A kernel that restructures the data to accommodate for the `cublasCgemmBatched()` function and replace the transpose thread in HASHPIPE. The array of pointers, `data`, is required as an input of the kernel and `data_restruc` is its output.

`signed char * data_in(char * input_filename)` - Reads the data file, and returns it as an array of pointers. The file name of the data file is required as an input.

`void beamform()` - Contains the `cublasCgemmBatched()` function that performs the beamforming operation. The `cublasCgemmBatched()` function is found in the cublas library and performs a matrix-matrix multiplications of an array of matrices.

`void sti_reduction(cuComplex * data_in, float * data_out)` - A kernel that performs the short time integration using a reduction algorithm commonly used on GPUs to reduce the number of threads per block. The kernel requires the output of the `beamform()` function as its input, `data_in`, and produces an output, `data_out`.

`void run_beamformer(signed char * data_in, float * data_out)` - This function calls the data restructure, beamform and sti reduction functions. The returned value of `data_in()` is required as its input, and its output is the short time integration kernel's output.