PST PI3 Report

This document summarizes some of the more significant work the PST team has done during program increment 3. The first section discusses the difference between SKA Mid and SKA Low PFB channelizers and their respective FIR filter coefficients. The second section discusses the work we've done towards completing feature SP-149.

SKA Mid FIR Filter Coefficients

Background

One of the Level 1 Requirements for SKA Mid, SKA1-CSP_ASS-405, states that "When Pulsar Timing on SKA1_Mid, the total spurious power, integrated over all time samples greater than 300 nanoseconds from the center of the narrowest possible impulse response function of the SKA1_Mid in that beam's observing Band, shall be less than -50 dB."

We put together a testing pipeline, written in Matlab, that attempts to determine if the novel FFT based PFB inversion algorithm used in the PST is capable of meeting this specification. With this pipeline we do the following:

- Generate a single channel, non dispersed, single bin impulse. The position of this impulse is parameterized.
- Channelize the impulse using an implementation of the Polyphase Filterbank algorithm, producing the number of channels supported by the FIR filter coefficients supplied to the PFB.
- Synthesize the channelized impulse using FFT based PFB inversion. This includes applying a Tukey FFT window on fine channels, in addition to appropriately rewinding through input data as the algorithm processes computational blocks.
- Compute purity metrics, namely the maximum spurious power and the total spurious power. Here, spurious power refers to any power found outside the 300 nanosecond region on either side of the original impulse position. If the FFT based PFB inversion algorithm were to perfectly reconstruct the original impulse, both these metrics would be zero.

We computed purity metrics for impulses placed strategically throughout the input domain. Given the size of input data vectors, and limited computational efficiency, we could not calculate temporal purity metrics for impulses placed at every point in the input domain. However, given the behaviour of the PFB inversion algorithm, especially when operating on data streams (instead of single blocks), we can predict where the algorithm will perform poorly.

SKA Mid and Low Performance Disparity

When calculating purity metrics using FIR filter coefficients designed for use with SKA Mid, we discovered that the performance of the PFB inversion algorithm was significantly worse than when doing corresponding tests for SKA Low. Moreover, we found that when we generated our own filter coefficients designed for SKA mid, performance increased significantly.

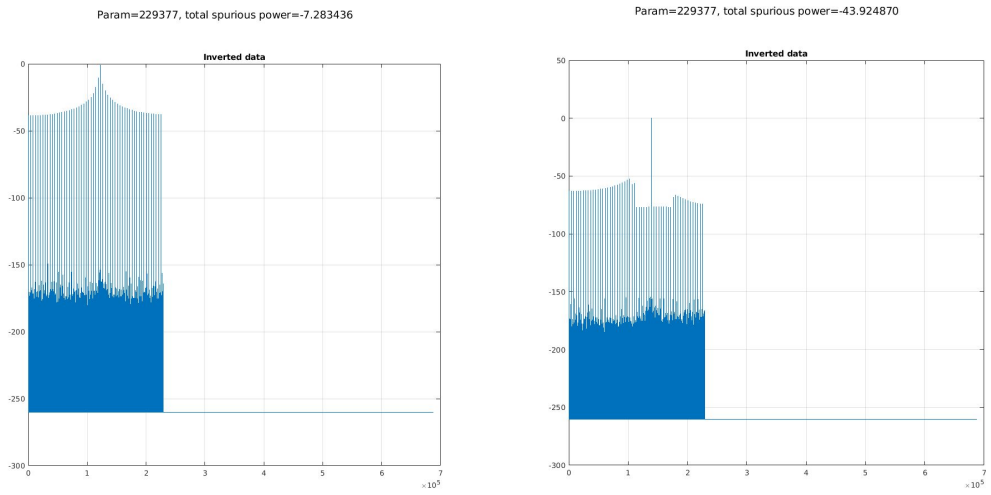

Figure 1.1 Results of PFB inversion for single impulse. Official SKA mid FIR filter coefficients on the left, unofficial coefficients on the right.

Figure 1.1 shows the dramatic difference between using these two sets of filter coefficients. We asked some of our colleagues who work more closely with SKA Mid filter design to help us understand why we were seeing this disparity. Luckily, one of our colleagues sent us the code they use to develop SKA mid FIR filter coefficients and the PFB inversion implementation they use to channelize data. After combing through this code, we discovered that not only are SKA Mid and Low FIR filter coefficients generated in a fundamentally different fashion, but the channelizers themselves differ in how they handle oversampling induced phase offsets. The following table summarizes some of these differences.

	SKA Low	SKA Mid
FIR Filter Coefficients: Design Stages	Uses a single stage approach. Given the relatively smaller number of taps per channel and output channels, SKA	Uses a two stage approach. This approach is necessary due to the high number of taps per channel, and the

	Low can afford to generate coefficients in a single stage.	large number of PFB output channels.[1]
FIR Filter Coefficients: Oversampling	Does not take into account PFB oversampling ratio.	Takes into account the PFB oversampling ratio. This is why SKA Mid has 24.5 filter taps per channel.
Polyphase Filterbank: zero-padding	No zero padding.	Pads the input with a number of zeros equal to the length of the FIR filter.
Polyphase Filterbank: FFT	Uses forward FFT.	Uses inverse FFT.[2]
Polyphase Filterbank: output circular shift	Does not apply circular shift.	Applies a circular shift to the channelized output. The delay is equal to half the length of the FIR filter divided by the number of output channels divided by the oversampling ratio.[3]

Table 1.1 Summary of differences between SKA Mid and SKA Low PFB channelizers and FIR filter coefficients

Each means of creating filter coefficients and channelizing works on its own. However, we cannot use filter coefficients generated with the SKA Low algorithm in conjunction with the SKA Mid channelizer, or vice versa.

In the PST_PFB_inversion_verification repository, these separate channelization approaches have been crystallized in the form of Matlab code:

- polyphase_analysis_padded.m: Performs SKA Mid channelization.
- polyphase_analysis.m: Performs SKA Low channelization
- design_PFB_FIR_filter_two_stage.m: Creates filter coefficients compatible with SKA Mid channelizer.
- design_PFB_FIR_filter.m: Creates filter coefficients compatible with SKA Low channelizer.

[1] This two stage approach uses “zero stuffing” to generate a sufficiently large FIR filter for the purposes of SKA mid. This works by designing a filter for a cutoff frequency that is some integer multiple larger than the intended cutoff frequency. This filter is then transformed into the frequency domain, and the resulting array is “stuffed” with enough zeros such that when transformed back into the time domain, it will cutoff at the desired frequency.
[2] Interestingly, if we switch to using a forward FFT, then we get significantly worse performance.
[3] If the length of the FIR filter is 196, the number of output channels is 8, and the oversampling ratio is 8/7, then the circular shift applied after channelization is equal to 14 (196 / 2 / (8 * 7 / 8).

InverseFilterbankEngineCUDA

The InverseFilterbankEngineCUDA is the name of the class in DSPSR that implements the FFT based PFB inversion algorithm, operating on the GPU. The main goal of PI3 has been designing, implementing, testing, and profiling this class.

Design

The InverseFilterbankEngineCUDA class is one of two “engines” in DSPSR that perform the PFB inversion algorithm on some channelized data. In the context of DSPSR, engines represent implementations of a given algorithm that are targeted to a specific hardware or software setup, or to a specific domain of input data, but united through a common interface. Under the hood, the CUDA and C++ implementations of the PFB inversion algorithm work quite differently, but from the perspective of the parent InverseFilterbank class, they are simply objects that operate on input data, saving the result in some output container. This approach allows for modular and extensible code, as it separates high level application logic from the nitty-gritty of algorithm implementation. While no plans exist at the moment for implementing more PFB inversion engines, we could imagine leveraging this architecture to create an OpenCL engine, or engines that are tuned to operating on data with large numbers of input channels.

Before implementing the InverseFilterbankEngineCUDA class, we spent time considering how to parallelize the PFB inversion algorithm before breaking the algorithm up into discrete computational units, with defined inputs and outputs. These units represent individual CUDA kernels. In the end, the InverseFilterbankEngineCUDA member function that actually performs the PFB inversion algorithm basically chains together the inputs and outputs of these kernels.

Figure 1.2 contains a visual representation of the kernels inside the InverseFilterbankEngineCUDA. Ovals represent kernels, and rectangles represent input or output arrays. Rectangles with dashed lines indicate that a kernel needs access to some static asset like a dedispersion response. Blue arrows indicate over which dimensions kernels operate. The long blue arrow on the right indicates that the entire transformation operates over the time dimension.

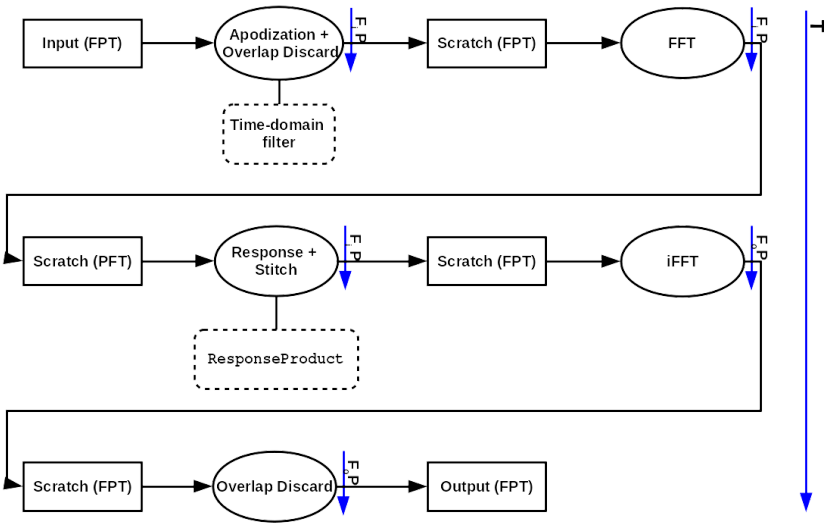

Figure 1.2 Graphic showing each of the CUDA kernels comprising the InverseFilterbankEngineCUDA.

Implementation and Testing

We adopted a test driven development approach to implementing the InverseFilterbankEngineCUDA transformation. This means that we wrote tests and implementation code roughly at the same time.

We chose to use the Catch2 unit testing framework to build a suite of unit and component tests. Catch2 is distributed as a single header file, meaning that it can easily be bundled up with DSPSR, circumventing the need to add any external dependencies. Tests are configurable via a TOML file, meaning that it is easy to add new test cases. This is particularly useful when switching between visually debugging small transformations and running transformations against a variety of input data sizes. tinytoml, a header only library, provides the interface to interacting with TOML files in DSPSR tests.

Before assembling kernels in the `InverseFilterbankEngineCUDA::perform` member function, we wrote static methods for each of the class's associated kernels. In addition, we wrote C++ versions of the kernels, used as reference implementations in the unit tests. We wrote test cases for each kernel (or rather, the static method that calls it), comparing its result to the reference implementation.

Once all the test cases for each of the kernels were passing, we created a component test that configures the InverseFilterbankEngineCUDA and then calls the member function that runs the PFB inversion algorithm. This required some boilerplate to get up and running, but it made implementing the actual algorithm much easier, as we didn't have to run the entire end-to-end DSPSR pipeline.

Validation

The most important aspect of the InverseFilterbankEngineCUDA is ensuring that it matches the InverseFilterbankEngineCPU to within floating point numerical accuracy. We created an integration test that runs both engines on the same input TimeSeries (filled with random numbers ranging from zero to one), and compares the respective outputs. In order to delve into what each engine was doing at each step of the transformation, we added an event emitter that dumps the internal state of the signal after specified points. In addition to being able to compare the output of the entire transformation, we could also compare the two engines at each step of the operation.

We used a C++ implementation of the numpy isclose function to test whether or not individual values are numerically close:

```
template<typename T>
bool isclose (T a, T b, float atol, float rtol)
{
  return abs (a - b) <= atol + rtol * abs(b);
}
```

For single precision floating point numbers, both `atol` and `rtol` should be 1e-5.

This integration test revealed an interesting discrepancy between the C++ and CUDA engines. As data size grew, so did the relative error between the implementations. In particular, we noted a significant jump in error after each of the engines performed FFTs. The following abridged testing logs illustrate this issue:

```
test_InverseFilterbankEngine_integration: fft_window 2097152/2097152 (100%)
test_InverseFilterbankEngine_integration: fft 2097087/2097152 (99.9969%)
test_InverseFilterbankEngine_integration: response_stitch 1572825/1572864
(99.9975%)
test_InverseFilterbankEngine_integration: ifft 1538338/1572864 (97.8049%)
```

Each row in this log shows show the percentage of values from each stage of the CPU and CUDA PFB inversion implementations for whom the `isclose` function returns true. Notice that the number of unequal values grows slightly after the forward FFT ("fft") before increasing dramatically after the inverse FFT ("ifft"). The number of unequal values decreases after the response_stitch stage because that stage of the algorithm removes the oversampled regions of input fine channel spectra.

We took a two pronged approach to investigating this issue. First, we created a unit test that compares the results of cufft plans and the fftw based plans created in psrchive. Second, we ran our Python purity tests to determine what, if any, effect the error would have on temporal purity. These Python purity tests can be found here.

The first test revealed that the number of nonequal values in the results of cufft and fftw FFTs are similar to that of the first forward FFT in the PFB inversion algorithm.

```
test_FTransform_cufft_precision: 32759/32768 (99.9725%)
```

Given that the PFB inversion algorithm requires two consecutive FFTs, it could be that FFT induced differences between the two implementations are accumulating. We need to investigate further to fully elucidate whether this is the root cause of the discrepancy between the two implementations.

The second test involves running the InverseFilterbank engines on a range of channelized time domain impulses, producing plots of temporal purity. Figure 1.3 shows the output of this test. Visually, it is difficult to discern any difference between the purity output of the CUDA or C++ implementations of the PFB inversion algorithm. Given that both engines produce similar plots of temporal purity, we are confident in the correctness of the InverseFilterbankEngineCUDA.

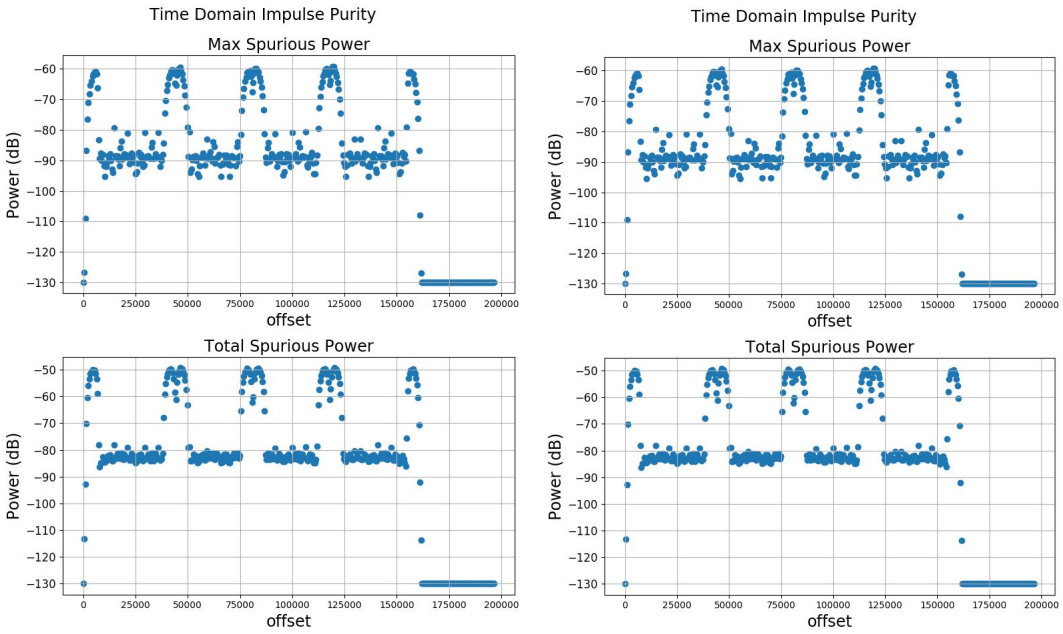

Figure 1.3 Impulse response of `InverseFilterbankEngineCUDA` (left) and `InverseFilterbankEngineCPU` (right)

Performance Testing

One part of the acceptance criteria for SP-149 (PFB Inversion Accelerated Prototype Implementation) states that the CUDA implementation of the PFB inversion algorithm should be able to perform in real time. This means that the implementation must be able to process data at the same rate or faster than the incoming data rate. The following table summarizes the data rate for the whole RF bandwidth of each of the SKA subbands[4]

Band	Low	1	2	3	4	5a	5b
RF Bandwidth (MHz)	300	700	810	1400	2380	2500	2500
Number of Input Channels	81000	13021	15067	26042	44271	46503	46503
Data Rate (Gb/s)	25.6	~51.2	~59.2	~76.8	~85.7	~91.4	~91.4

Table 1.2 Summary of the incoming PST data rate for each subband of the SKA

[4] See SKA-TEL-CSP-0000090_4_PSTDetailedDesign_Jameson_2018-10-25.pdf

We created a benchmarking executable that determines the data processing capability of the InverseFilterbankEngineCPU and InverseFilterbankEngineCUDA. This benchmark populates input TimeSeries objects with random numbers ranging from zero to one. Table 1.3 shows the result of running this benchmark on input data arrays of varying sizes. All tests were run with dual polarization data.

dspsr-bench is a Python package that runs the DSPSR benchmarking executable on range of data shapes. We used it to produce the data in table 1.3.

Input Channels	Output Channels	Input FFT length [5]	Data Rate (Gb/s) — CUDA	Data Rate (Gb/s) — C++
8192	1	1024	216.677	1.479
8192	16	1024	256.012	2.455
8192	256	1024	259.28	3.258
8192	1	2048	214.72	1.442
8192	16	2048	246.469	2.392
8192	256	2048	260.113	3.1
32768	1	1024	211.991	1.391
32768	16	1024	240.285	2.31
32768	256	1024	261.314	2.868
32768	1	2048	207.609	1.194
32768	16	2048	237.12	1.994
32768	256	2048	260.889	2.651

Table 1.3 Data rate of the CUDA and C++ InverseFilterbank engines for a range of data shapes.

Table 1.3 suggests that the CUDA implementation of the PFB inversion algorithm will be able to process SKA data in real time. These findings are by no means indicative of the performance of the entire PST subelement. Assessing the performance of the entire PST subelement would necessarily have to account for additional processing elements, like RFI mitigation as well as practical limitations like data transfer rates.

[5] The block size of the PFB inversion algorithm is determined by the size of the forward FFT that is applied to coarse channel time domain data. Larger input FFT lengths are desirable in that they improve the impulse response of the algorithm.

Profiling and Optimization

We used the Nvidia Visual Profiler (nvvp) to profile the benchmarking executable with the idea of identifying optimization opportunities that would be relatively easy to implement. This exercise proved useful, in that we were able to reduce the execution time of the three custom kernels that comprise the InverseFilterbankEngineCUDA by a factor of two or three. These optimizations came at the cost of reduced flexibility -- the custom kernels are no longer able to operate across multiple time or fine channel blocks. Figures 1.5 shows the occupancy for each of the custom kernels in the InverseFilterbankEngineCUDA. High occupancy means that the kernels are utilizing GPU resources well. We iteratively adjusted our kernels in order to increase occupancy and decrease execution time, ensuring that unit and component tests passed after each change in source code.

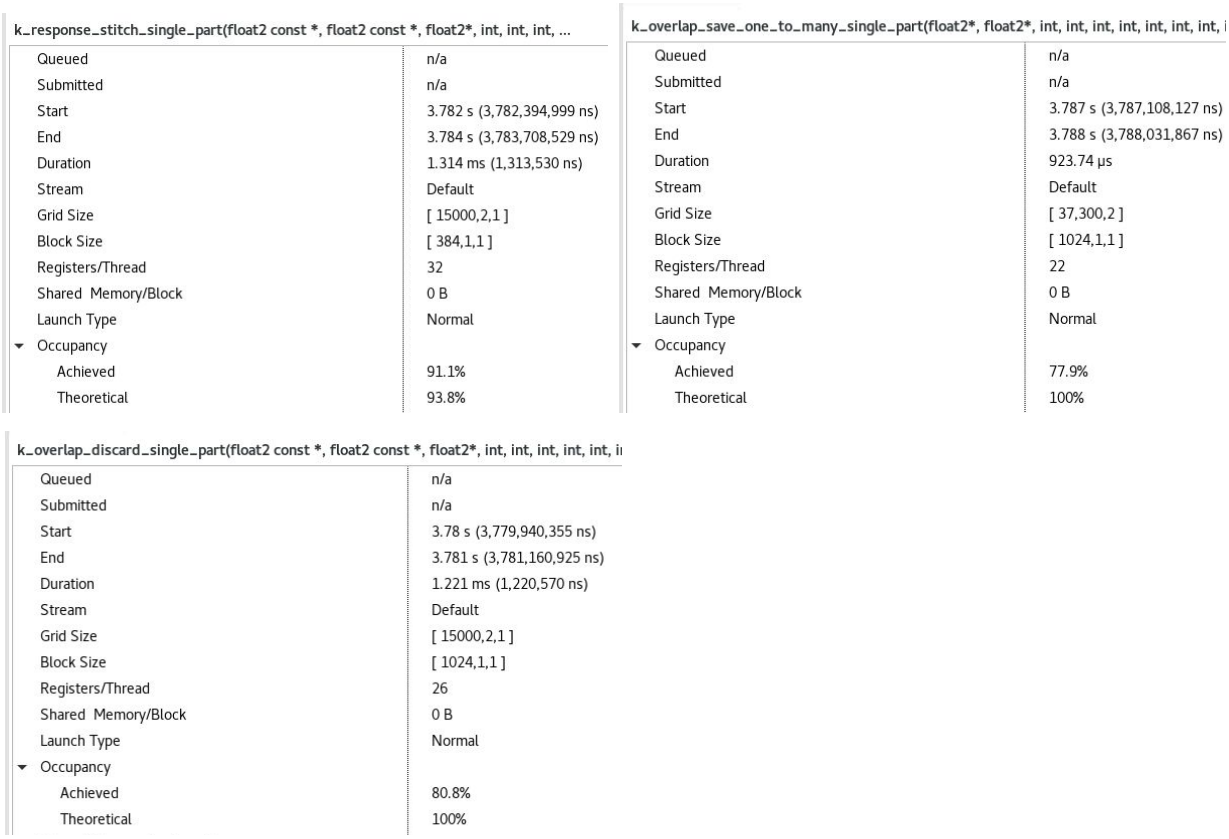

Figure 1.5 Occupancy reports for each of the custom kernels in the InverseFilterbankEngineCUDA

We identified a few key areas that could potentially increase the performance of the implementation:

- Operate across multiple time or fine channel blocks at the same time. Right now, the InverseFilterbankEngineCUDA is parallelized in frequency and polarization, but still iterates over processing blocks. This optimization would dramatically increase the memory usage of the implementation, potentially making it unwieldy for large data sizes.
- Implement cuFFT callbacks. cuFFT callbacks provide a mechanism by which we could reduce the total number of trips from global memory. This would be especially beneficial in the case of the InverseFilterbankEngineCUDA, as the three custom kernels it uses spend most of their time reading and writing from and to global memory.