CUDA Based Polyphase Filter

Mark McCurry

***Abstract*—This paper presents the evaluation of the use of a graphics processor for realtime radio astronomy DSP (Digital Signal Processing) within VLBI (Very Long Baseline Interferometry). A polyphase filter bank (pfb) was implemented in a prototype application to convert external ADC input into channelized frequency streams. This system was tested with a 32 channel pfb, 8 bit samples, and 8 taps/channel. With a prototype system, 512 Mega-samples/second could be easily processed and 890 Mega-samples/second is possible. Instruction throughput is the current limitation, so a modest increase in the graphics card's processing speed will permit the desired speed of 1024 Mega-samples/second. This makes GPUs an interesting candidate for a cost effective upgrade as both software and hardware systems progress.**

***Index Terms*—Polyphase Filter Bank, GPU, CUDA, CUFFT, radio astronomy, VLBI**

I. Introduction

WITHIN the field of radio astronomy, upgrading existing equipment to get performance and cost gains occurs often. For VLBI (Very Long Baseline Interferometry), the implementation of PFBs (Polyphase Filter Banks) has been done on FPGA (Field Programmable Gate Arrays) boards in order to meet the required data rates. As GPU (Graphics Processor) hardware has become cheaper and faster with time, it has started becoming an option for this signal processing. This project is an investigation into the performance of a GPU as a replacement for the DSP (Digital Signal Processing) performed by the FPGA boards.

A. Background

Polyphase filters for radio astronomy have gone through several iterations. Initial filters were implemented as analog bandpass banks. Later they shifted over to the FPGA centric RDBE (Roach Digital Back-End). Now with cheaper access to GPU based processors, they may be the next evolutionary step. As indicated by [1], the Nvidia GPUs can be a cost effective general purpose computing hardware for this task.

1) General Terminology: In order to effectively read through this report, some basic terms for signal processing and the surrounding components should be known. The built system is designed to work on a stream of samples, or instantaneous values of the signal, to decompose its frequency domain information. Each one of these samples is taken at a regular rate called the sampling rate. This processing is done on a set number of frequency channels, or ranges for frequency information to be grouped in.

On both ends of this system work is done to transform between fixed and floating point numbers. Floating point numbers are like numbers that have a set number of significant digits and are scaled as needed with an exponent. Fixed point numbers have a set range and no exponent for varying their scale.

These fixed point samples are passed to the system with UDP packets. After removing the packet header, a frame of data is gathered. This can be grouped with other frames to form a chunk of data, which is the basic processing unit of the designed system.

2) FIR Filters: One of the major components needed for the PFB is a polyphase FIR filter. Ignoring the polyphase aspect, a FIR filter is a Finite Impulse Response filter, or more simply, a means to change the frequency contents of an input signal. Every sample output from this system is the sum of scaled previous components. A typical way to write this is in a difference equation, which describes the output via previous samples. One of the most simple examples is the running average filter. For a running average of three input samples x, the output y at sample i is defined in Equation 1. This has a frequency response shown in Figure 1

$$y[n] = \frac{1}{3}x[n] + \frac{1}{3}x[n-1] + \frac{1}{3}x[n-2] \tag{1}$$

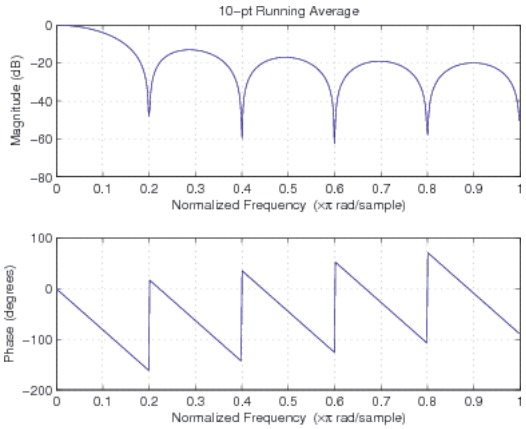

Fig. 1. 10 Point Running Average Filter Response

In the multirate case, some portions of the signal processing runs at a different sampling rate than the rest of the system. In the case of polyphase filters, all the samples get processed, but with different paths. The specifics of the used filter are discussed later.

B. Goals

This project was intended to demonstrate that existing hardware could replicate and existing FPGA based DSP system. This system was designed to have an input rate of 1024

REU program, MIT Haystack, Westford, MA

Fig. 2. System block diagram

Mega-samples/second or 1024MB/s when ignoring protocol overhead. The intended PFB would perform 32 point FFTs and it would use a 256pt prototype FIR filter. The output would be a series of 4bit complex samples, with 2bits for each real and imaginary component. The PFB is based upon a sample MATLAB implementation discussed in the next section.

II. Methods

A. Prototype

As a reference implementation, a series of Matlab scripts were provided. These Matlab scripts are provided in the appendix for comparison. These files were used to produce the test filters, which the final implementation conforms to.

B. Equipment

The testing setup was composed of a desktop computer, RDBE, signal generator, clock generator, and 1pps generator. The desktop computer has one 64-bit core, an standard ethernet NIC for RDBE setup, a 10GbE NIC for RDBE data transfers, a Nvidia Tesla C2050, and stock components.

1) RDBE: The roach board for this project was a source of external vdiff packets. The Roach board was designed by CASPER[1] and produced by DigiCom[2].

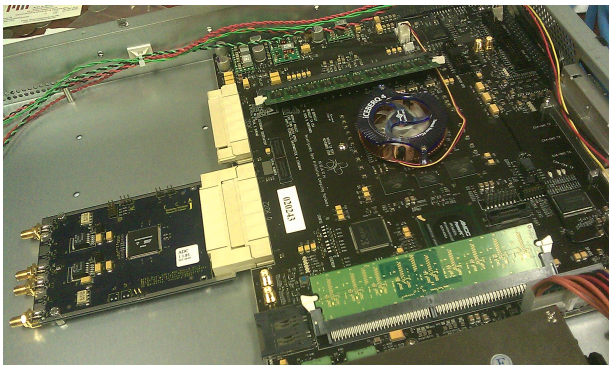

Fig. 3. ROACH (Reconfigurable Open Architecture Computing Hardware)

2) NIC: The network interface used to interface with the roach was an Intel 82598EB 10GbE card. This was placed in a PCI-E x4 slot, limiting the overall data rate to an absolute maximum of 800 MB/s unidirectional. As this is below the desired 1 GB/s, full speed cannot be tested in the current setup.

3) Tesla GPU: A Nvidia Tesla graphics card was used. This card is one of the high end scientific computing cards offered by Nvidia. It offers a significant amount of cores, memory, and speed when compared to other cards. Using this card the computational prices for various operations could be explored. If possible this card could be replaced with another to reduce costs. The specifications for this card are summarized in Figure 9.

[1]Center for Astronomy Signal Processing and Electronics Research casper.berkeley.edu/group.php

[2]http://digicom.org/special-products/roach-board.html

Fig. 4. Nvidia Tesla Graphics Card

4) Supporting Hardware: Several other instrument were used to support the roach board. A one pulse per second generator was used to initialize the system. A clock generator was used to set the sampling rate of the board. A function generator was used to provide test signals to the system.

C. Code

The code for this experiment was a combination of networking and CUDA (Compute Unified Device Architecture) code written in C++. All code was built and tested under CUDA 4.0.

For the main loop of the program can be summarized in Algorithm 1.

Algorithm 1 Main loop algorithm

```
SetupFilter
while processing do
  Synchronize(pfb.stream)
  Save(Result) when Result
  Receive(packets)
  Process(packets, pfb)
end while
```

1) Filter Parameters:

$$\operatorname{sinc}(t) \equiv \frac{\sin(\pi t)}{\pi t} \tag{2}$$

$$\texttt{buffer}[i] = F_c \operatorname{sinc}([)\, F_c(i - \texttt{taps}/2)] \tag{3}$$

Where taps is the total number of taps, F_c is the cut-off frequency determined to be $\texttt{channels}^{-1}$, and $i \in \{\mathbb{Z} | 0 < i \leq \texttt{taps}\}$.

2) Convolution: For the PFB, convolution is performed on the decimated filter and decimated samples. For interleaved samples, N channels, and M taps per channel, the definition of the output is:

$$y[i] = \sum_{i=0}^{M} x[i * N] * coeff[i * N] \tag{4}$$

In the past this operation was done using fixed point arithmetic, which is possible with CUDA. Fixed point is not used, as under current graphics cards (Compute Capability 2.0), fixed point operations take twice as long. On the next compute capability, 2.1, fixed point operations will take three times as long[2].

3) CUFFT: After the convolution, CUFFT[3] is used to perform the forward FFT on the samples. In order for the library to perform the R2C transform, it requires extra padding, as shown in Figure 5. This padding could be prevented in two

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	X	X

Fig. 5. Before/After Frame Packing

main ways. Samples could be interleaved in place to allow for a C2C transform, later recovering the real portion of the data. This option could yield subpar performance if special care is not taken to perform sequential reads and writes[3].

It would also be possible to write a custom FFT (Fast Fourier Transform) routine to perform the operations in place, dropping the unwanted channels. This option has the issue that a custom FFT is likely not as optimal as the existing one and it would require extra development time.

4) Quantization: With this done, the samples are quantized and optionally stripped of unused frequency channels. Within VLBI, the conventional bits per complex sample are 4 bits as shown in [4]. The quantization factor was found based upon insuring the distribution documented in [4].

III. Results

A. Filter Performance

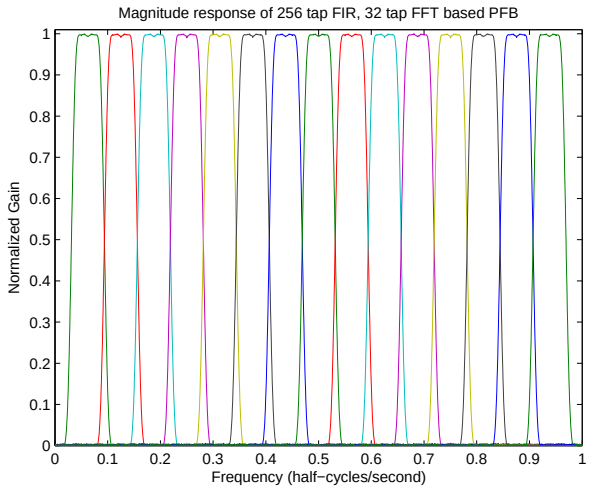

Fig. 6. Acheived Frequency spectrum of desired bands

In order to test the filter performance to verify all algorithms within the GPU, a series of tests were run. For both floating point and fixed point stages it was verified that all functioned as expected. A tone centered in each channel resulted in $30x$ or more signal in that channel than adjacent channels.

In order to further see the performance of the system, dumps of the spectrum were made and verified to hold true with an external signal generator. The resulting spectrum generated by `spectrum-summary` can be seen in Figure 6, which is identical to Figure 7 with the removal of DC. This and the test suite can be found with the source at http://fundamental-code.com/gitweb/?p=gpfb.git or by cloning a copy with: `git clone git://fundamental-code.com/gpfb.git`

[3]CUDA FFT library[3]

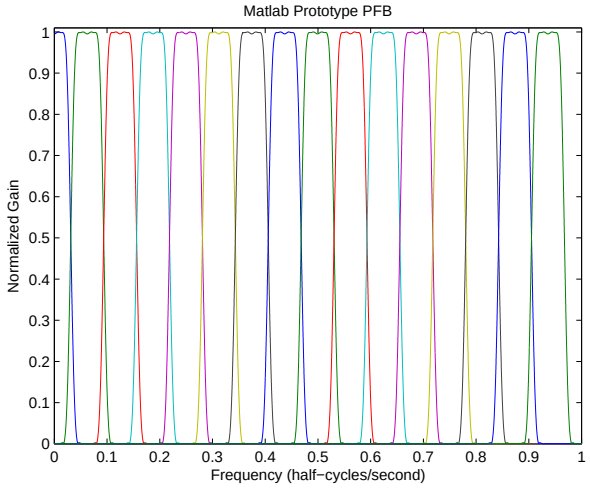

Fig. 7. Prototype Frequency spectrum

B. Real Time Performance

This system was tested at half the data rate, 512 Mega-samples/s, as the full data rate was not supported by the data source or the used NIC. At the time of writing, the desired full data rate with the specified filter parameters and desired output format was not accomplished. This desired format involved stripping out unwanted channels that would not be used in the correlator. With both of these constraints, the system is able to run at half speed with a margin of error. When sacrificing the output format, half speed operations run without problem in realtime. The sacrified format would include all output channels fo the system, include those that were aliased, ie. the DC and highest frequency channel.

In order to test the system further, it was tested on a GTX470, which is about $\frac{1}{10}$x the cost of the used Tesla card. As shown in Table I, this could be a much cheaper option for lower bandwidth systems.

All tests were averaged over 1024 times and were done with 2 parallel executions of the pfb. This was done with 256 taps on the protype filter and 32 channels in the system. The size of a chunk was 64,000,000 samples. All tests that did not involve the use of the RDBE packets tracked only the gpu processing time with IO with respect to CPU/GPU memory. One execution of the pfb is shown in Figure 8.

IV. Future Work

In order for this project further, several things need to be addressed:

- Software cannot run at full speed

TABLE I
PERFORMANCE OF GPU CODE[4]

Performance Metrics	Data Input Rate
Reference Implementaiton	744 MB/s
No extra channels	540 MB/s
Hardcoding FIR size	756 MB/s
Hardcoded FIR, Hand Tuned Block Size	890 MB/s
Using 1/10 cost 470GTX	637 MB/s

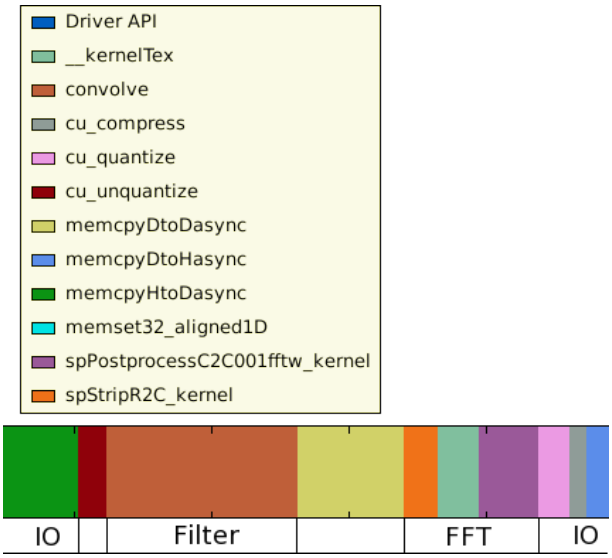

Fig. 8. Display of One Iteration from Computeprof

- Hardware cannot run at full speed
- Software needs enhanced configuration support
- Software parameter space has not been fully explored

A. Full Speed SW

As mentioned in the results, the overall software performance is 74% of the desired speed. It is suspected that further gains can be made here, but the exact way to make further gains is not clear, but likely related to CUDA specific semantics. The data rates are near those cited in [1], indicating that the implementation is not unreasonably slow.

B. Full Speed HW

There are currently two issues that prevent the hardware from reaching the desired 1024 MB/s input data rate. The first is the hardware configured for this REU does not support the output rate. This will need to be modified by one of the resident computer engineers. The second is that the 10Gb NIC can not currently support the data rate. This can be fixed by moving it to a x8 PCI slot, which is not available in the current testing machine.

C. Configuration

At this time most of the parameters for the system are coded directly into the .cpp/.h files. This means that the behavior of the utilities are determined at compile time. Ideally the program would be parameterized with an external configuration file. One candidate for this is `libconfig`[5]. This offers a simple file format that could hold all parameters.

D. Parameter Space Optimization

At this point, the prototype has not had the relationship between filter taps, and channels related back to the overall data rate. In the end, the overall goal is to optimize the signal-to-noise ratio of the system, which will vary as the FIR becomes longer or shorter. It is suspected that a reduction of the filter taps to increase the sample rate will improve the signal-to-noise ratio the most.

V. CONCLUSIONS

As a result of this examination, it appears that GPUs will be an inportant option relative to the RDBE system in performing DSP with radio astronomy. Although the deomonstrated data rates near 900 Mega-samples per second indicate that it does process data in the right order of magnitude for this system to becomboe a vialble extention as GPUs continue to advance.

REFERENCES

[1] K. van der Veldt, "A polyphase filter for gpus and multi-core processors," Master's thesis, Universiteit van Amsterdam, April 2011.
[2] Nvidia, *NVIDIA CUDA C Programming Guide*, 2011.
[3] Nvidia, *CUDA CUFFT Library*, 2011.
[4] W. Brisken, "Complex sampling considerations," 8th US VLBI Meeting, 2010.
[5] Nvidia, *CUDA C Best Practices Guide*, 2011.

APPENDIX

This project is designed to provided an implementation of a polyphase filter for use with vdiff packets and a software correlator. Requirements:

- cmake
- CUFFT
- CUDA

From the current directory:

```
mkdir build
cd build
cmake ..
make
make test
```

Currently few of the utilities process command arguments or configuration files. Module dependency is fairly minimal, so changing this should require fairly minimal work.

A. Full Processing(pfb)

This program executes the full processing chain and dumps the result to file for analysis. As this has not be used with tools further in the signal processing chain, the output is a csv of channels by time. This process occurs with full quantization.

[5]http://www.hyperrealm.com/libconfig/

B. Realtime Performance(rt-summary)

Summarizes the realtime performance of the GPU processing. This summary does not account for any overhead that CPU based vdiff packet work may involve, but this has tended to be fairly minimal.

C. Filter Characteristics(spectrum-summary)

Produce the frequency response of all channels in the filter. This is performed with floating point data. For basic information on fixed point responses, see the fixed point filter test.

D. Packet Viewing(pkt-dump)

Viewing the packet output of the roach system can be an important stage of debugging. This program accepts the number of packets as an argument.

As mentioned previously, the parameters of the system currently reside in the code, some more formally than others. For the entire system, the characteristics of the pfb are set in **params.h**

- CHANNELS: The total number of channels or split FIR sections for the PFB. This sets the output to be (CHANNELS/2+1) frequency bins.
- TAPS: The number of taps for the prototype FIR filter. It is assumed that TAPS%CHANNELS = 0.
- FS: For testing FS is used to describe frequency over normalized frequency

For specific utilities:

- MEM_SIZE: Deprecated buffer length for testing, defined in **param.h** used in frequency response tests and spectrum summary
- TEST_LENGTH: Number of iterations to average the performance over. Defined for **realtime.cpp**, only for rt-summary.
- Packets: This defines the number of packets to be read in **main.cpp**. This is also defined separately in **realtime.cpp**.
- Addr: The current address of the source of vdiff packets. Defined and used in **rdbe.cpp**.
- Port: The current port of the source of the vdiff packets. Defined and used in **rdbe.cpp**.

../FIRsimulation.m

```
close all;

N = 50; %number of frames
fs = 1024; %sample rate (MHz)
dec = 8; %decimation factor
taps = dec*8; %number of taps
fo = 12; %test tone frequency (MHz)

Nt = N*taps; %total number of ADC samples
t = 0:1/fs:((Nt-1)/fs); %ADC time sample array (us)
td = t(1:dec:Nt); %decimated time sample array

chbw = fs/dec;

s = cos(2*pi*fo*t); %test tone signalplot
%ch = 5;
%s = chirp(t,chbw*(-1/2+ch),t(end),chbw*(1/2+ch));

%remove the clock offset so that the input data samples are time aligned
%with the FIR stage output samples
% s = reshape(circshift(InputData',20)',1,numel(InputData));


%calculate FIR coefficients and reshape them into an
%'dec' by 'tap/dec' dimension array for polyphase processing
%this is the polyphase decomposition of the prototype filter
firc = firl(taps-1,1/dec);

%display filter response
figure(2)
freqz(firc,1)

firc = firc/max(firc);
cofs = fliplr(reshape(firc,dec,taps/dec));
figure(101)
plot(cofs)
figure(4)
freqz(cofs(:,1),1)

%decimate/filter input signal
sf = pfbconv(s,cofs);

%transform filtered signals to time domain
SF = fft(sf,[],1);
figure(99)
plot(rot90(sf))

figure(1),
plot(t(1:dec:end),sf(1,:))
mx = max(abs(SF(1)),abs(SF(2)));
figure(3),
for n = 1:(dec)
    subplot(dec,1,n)
    plot(real(SF(n,:)))
    axis([-inf inf -20 20])
end
```

../pfbconv.m

```
function sf = pfbconv(s,h)
%h: dec x (tap/dec)
%s: 1 x Ns
Nc = size(h,2); %number of coefficients per tap
Np = size(h,1); %number of taps
Ns = length(s); %length of input samples

sf = zeros(Np,Ns/Np);

for m = 1:length(s) %'m' is the full-rate (i.e. fast) time index

        %polyphase filter bank index for the current fast-time index
        n = mod(m-1,Np)+1;

        %decimated time sample index
        k = floor((m-1)/Np)+1;

        %signal sample indicies to apply the filter
        %there are always 'taps/dec' number of indicies
        sidx = (m-(Nc-1)*Np):Np:m;

        %discriminate sample indicies outside the index bounds
        %in general 'fidx' will be 1:(taps/dec) unless filter is
        %to be placed near the boundary samples of 's'.
        fidx = find((sidx>0)&(sidx<=Ns));
        sidx = sidx(fidx);

        %convolve the input samples with the filter coefficients
        sf(n,k) = sum(s(sidx).*h(n,fliplr(fidx)));

end
end
```

```
/usr/local/cudasdk/C/bin/linux/release/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Found 1 CUDA Capable device(s)

Device 0: "Tesla C2050 / C2070"
  CUDA Driver Version / Runtime Version          4.0 / 4.0
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 2687 MBytes (2817982464 bytes)
  (14) Multiprocessors x (32) CUDA Cores/MP:     448 CUDA Cores
  GPU Clock Speed:                               1.15 GHz
  Memory Clock rate:                             1500.00 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535),
                                                 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048,
                                                 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and execution:                 Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                Yes
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           3 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
       simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 4.0,
             CUDA Runtime Version = 4.0, NumDevs = 1,
             Device = Tesla C2050 / C2070
```

Fig. 9. Device Summary from Nvidia's deviceQuery

../PFBfilterresponse.m

```
fs = 1024; %sample rate (MHz)
dec = 32; %decimation factor
taps = 2^8; %number of taps

%time array
t = 0:(1/fs):((1/fs)*(taps-1));

%Polyphase prototype filter
s = firl(taps-1,1/dec);

%plot the prototype filter
figure(3), plot(t,s)

%zero pad prototype filter up by factor of 100 and calc spectrum
S = fftshift(fft((s),100*taps));

%frequency array of spectrum
f = ((-floor(100*taps/2):(ceil(100*taps/2)-1))/(100*taps))/(t(2)-t(1));

%plot the spectrum
figure(1),plot(f,20*log10(abs(S)))
% axis([-16 16 -0 1.1])

figure(2), plot(f,angle(S.*exp(1i*2*pi*f*(127.5/1024)))*180/pi);
axis([-16 16 -180 180])

%write spectrum to file
% dlmwrite('PFB32response.txt',[f' real(S)' imag(S)'],'newline','unix');
Sps = S.*exp(1i*2*pi*f*(127.5/1024));
Sps(angle(Sps) ~= 0) = abs(Sps(angle(Sps) ~= 0));

St = zeros(16,length(Sps));
for n = 1:16
    St(n,:) = circshift(Sps',(n-1)*floor(dec/(f(2)-f(1))))';
end

St = St/max(max(St)); %normalize
figure(3),
plot(f/512,St);axis([0 1 0 1.01])
xlabel('Frequency (half-cycles/second)')
ylabel('Normalized Gain')
title('Matlab Prototype PFB')
%grid

figure(4),
plot(f,angle(St),'linewidth',5);axis([0 512 -10 10])
xlabel('Frequency (MHz)')
ylabel('Phase (\theta)')
%grid
```