2010 CASPER Workshop

Tutorial 5: Wideband Spectrometer

Author: **Terry Filiba**

Version: **August 2010, v1.0**

Expected completion time: 2hrs

Contents:

Introduction

In this tutorial we will record data from a 400MHz bandwidth, 1024 channel spectrometer on the ROACH and further channelize that data in software using a GPU. For more information in spectrometers please refer to Tutorial 3 which will walk you through a complete spectrometer design.

The tutorial is split into 2 directories. The model directory contains the simulink model file and a precompiled bof file. The src directory contains a python script, gpu_spec_init.py, to initialize the ROACH and record some data to the data directory, a c file, gpu_fft.c, that will open file containing recorded data, fft it on the gpu, and record it back to the data directory, and another python file, plot_gpu_spectrum.py, to plot that result.

You should also have installed python, iPython, corr, aipy, numpy, pylab, and CUDA. As far as hardware goes, you'll need:

- a ROACH board;
- an iADC, which should be connected to *ZDOK0* on the ROACH; and
- a clock source, such as a signal generator, which should be connected to *clk_i* on the iADC.

- access to a computer with an Nvidia GPU and CUDA installed

PART ONE

ROACH Channelizer

Simulink Design Overview

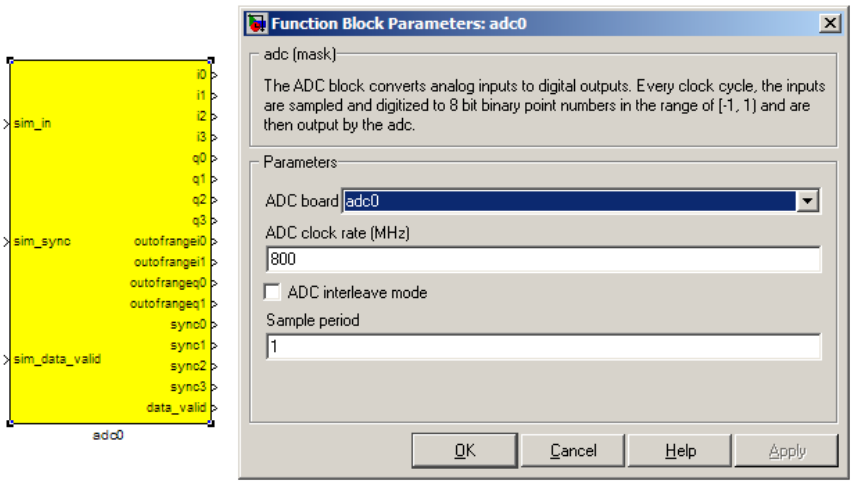
The best way to understand fully is to follow the arrows, go through what each block is doing and make sure you know why each step is done. To help you through, there's some “blockumentation” in the appendix, which should (hopefully) answer all questions you may have. A brief rundown before you get down and dirty:

- The **System Generator** block
- In the **MSSGE block**, the hardware type is set to 'ROACH:sx95t” and system is clocked at 200MHz using adc0_clk
- The sync generator which calculates the sync period based on the fft and accumulation parameters. The accumulation length is set to 4096 since we will record 4096 samples at a time from a single channel.
- The signal comes in through the **adc** block which generates 4 samples on each clock
- The samples from the **adc** are fed through the **pfb_fir_real** and **fft_wideband_real**.
- The data out of the **fft_wideband_real** is fed into a mux allowing us to select an odd or even channel based on the bottom bit of the channel_select register.
- The channel_select register is also compared against the channel_counter to tell us when we have data from the channel we want.
- Two shared brams **re_channel_bram** and **im_channel_bram** are used to record 4096 samples from a single channel.
- Below the channel selection logic there are also some scopes. These will continuously record data from the fft allowing us to plot the entire spectrum and ensure the roach is set up properly.

Without further ado, open up the model file and start clicking on things, referring the blockumentation as you go.

ADC

http://casper.berkeley.edu/wiki/Adc



The first step to creating a frequency spectrum is to digitize the signal. This is done with an ADC – an Analogue to Digital Converter. In Simulink, the ADC daughter board is represented by a yellow block. Work through the "iADC tutorial" if you're not familiar with the iADC card.

The ADC block converts analog inputs to digital outputs. Every clock cycle, the inputs are sampled and digitized to 8 bit binary point numbers in the range of [-1, 1) and are then output by the ADC.

The ADC has to be clocked to four times that of the FPGA clock. In this design the ADC is clocked to 800MHz, so the ROACH will be clocked to 200MHz[1]. This gives us a bandwidth of 400MHz, as Nyquist sampling requires two samples (or more) each second.

This block was created by Pierre Yves Droz. Further documentation can be found online and is courtesy of Ben Blackman.

INPUTS

sim_in	Input for simulated data. It's useful to connect up a simulink source, such as "band-limited white noise" or a sine wave.
sim_sync	Simulated sync pulse input. In this design, we've connected up a constant with value '1'.
sim_data_valid	Can be set to either 0 (not valid) or 1 (valid).

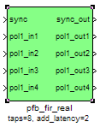
OUTPUTS

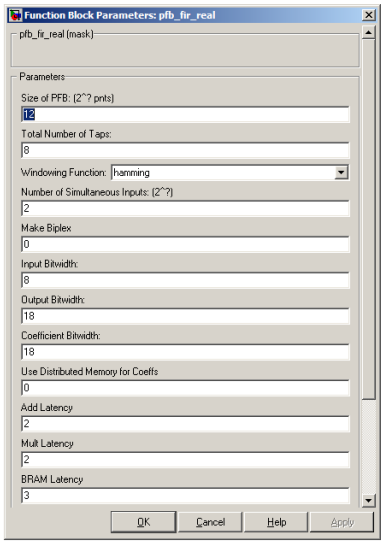
The ADC outputs two main signals: *i* and *q*, which correspond to the coaxial inputs of the ADC board. In this tutorial, we'll only be using input *i.* As the ADC runs at 4x the FPGA rate, there are four parallel time sampled outputs: *i0, i1, i2* and *i3.*These outputs are 8.7 bit.

1 In the XSG core config block, we have selected *adc0_clk* as our clock, which is provided by the ADC daughter card plugged into connector ZDOK0.

PFB_FIR_REAL

http://casper.berkeley.edu/wiki/Pfb_fir_real





There are two main blocks required for a polyphase filter bank. The first is the **pfb_fir_real** block, which divides the signal into parallel 'taps' then applies finite impulse response filters (FIR). The output of this block is still a time-domain signal. When combined with the **FFT_wideband_real** block, this constitutes a polyphase filterbank.

This block was created by Henry Chen, and most of the documentation presented here is courtesy of Ben Blackman.

INPUTS/OUTPUTS

Port	***Data Type***	***Description***
sync	*bool*	A sync pulse should be connected here (see iADC tutorial).
pol1_in1 *pol1_in2* *pol1_in3* *pol1_in4*	*inherited*	The (real) time-domain stream(s). As the ADC has four parallel time sampled outputs: *i0, i1, i2* and *i3,* we need four parallel inputs for this PFB implementation.

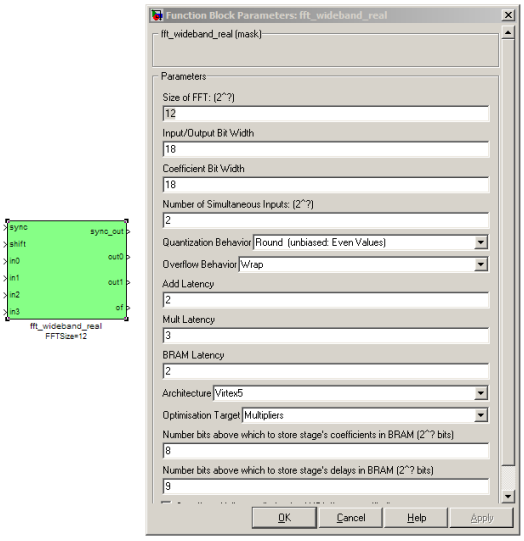
PARAMETERS

Size of PFB	How many points the FFT will have. The number of frequency channels will be half this. We've selected 2^{10}=1024 points, so we'll have a 2^{9}=512 channel filter bank.
Number of taps	The number of taps in the PFB FIR filter. Each tap uses 2 real multiplier cores and requires buffering the real and imaginary streams for $2^{PFBSize}$ samples. Generally, more taps means less inter-channel spectral leakage, but more logic is used. There are diminishing returns after about 8 taps or so.
Windowing function	Which windowing function to use (this allows trading passband ripple for steepness of rolloff, etc). Hamming is the default and best for most purposes.
Number of Simultaneous Inputs ($2^{?}$)	The number of parallel time samples which are presented to the FFT core each clock. The number of output ports are set to this same value. We have four inputs from the ADC, so set this to 2.
Make biplex	0 (not making it biplex) is default. Double up the inputs to match with a biplex FFT.
Input bitwidth.	The number of bits in each real and imaginary sample input to the PFB. The ADC

	outputs 8.7 bit data, so the input bitwidth should be set to 8 in our design.
Output bitwidth	The number of bits in each real and imaginary sample output from the PFB. This should match the bit width in the FFT that follows. 18 bits is recommended for the ROACH.
Coefficient bitwidth	The amount of bits each windowing function uses. The number of bits in each coefficient. This is usually chosen to match the input bit width.
Use dist mem for coeffients	Store the FIR coefficients in distributed memory (if = 1). Otherwise, BRAMs are used to hold the coefficients. 0 (not using distributed memory) is default
Latency	There's normally no reason to change this unless you're having troubles fitting the design into the fabric.
Quantization Behavior	Specifies the rounding behavior used at the end of each butterfly computation to return to the number of bits specified above.
Bin Width Scaling	PFBs give enhanced control over the width of frequency channels. By adjusting this parameter, you can scale bins to be wider (for values > 1) or narrower (for values < 1).

FFT_WIDEBAND_REAL

http://casper.berkeley.edu/wiki/Fft_wideband_real



The **FFT_wideband_real** block is the most important part of the design to understand. The cool green of the FFT block hides the complex and confusing FFT butterfly biplex algorithms that are under the hood. You do need to have a working knowledge of it though, so I recommend reading Chapter 8 and Chapter 12 of Smith's free online DSP guide (at *http://www.dspguide.com/*).

Parts of the documentation below are taken from the documentation by Aaron Parsons and Andrew Martens on the CASPER wiki (at http://casper.berkeley.edu/wik).

INPUTS/OUTPUTS

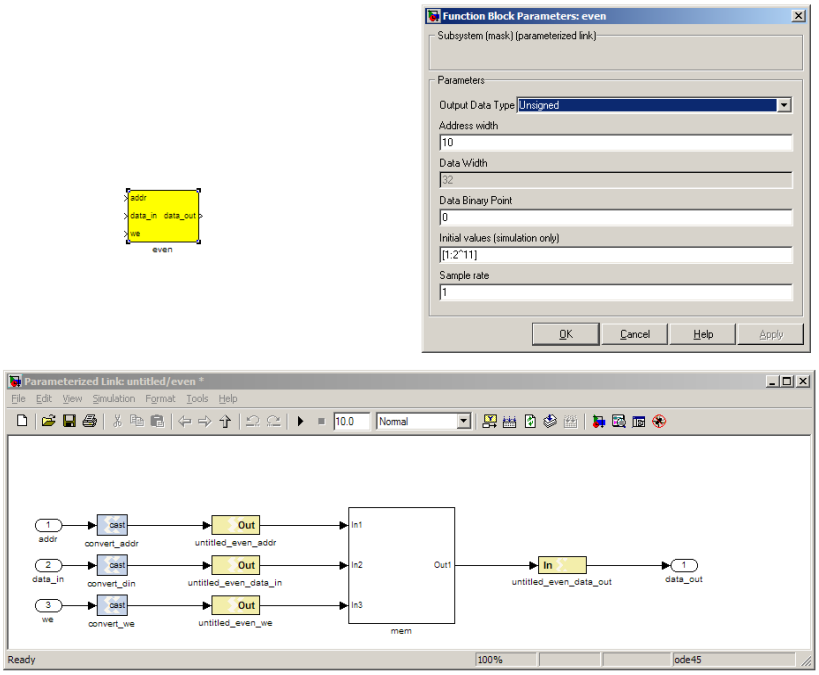
sync	Like many of the blocks, the FFT needs a heartbeat to keep it sync'd
Shift	Sets the shifting schedule through the FFT. Bit 0 specifies the behavior of stage 0, bit 1 of stage 1, and so on. If a stage is set to shift (with bit = 1), then every sample is divided by 2 at the output of that stage. In this design, we've set Shift to $2^{13-1}-1$, which will shift the data by 1 on every stage to prevent overflows.
in0 ***in1*** ***in2*** ***in3***	Four inputs for the parallel data streams coming from the ADC, through the pfb_fir_real filter block, and into here. Just connect them up.
out0 ***out1***	The FFT produces two signals, the real part (**out0**, cosine wave values) and the imaginary part (**out1**, sine wave values). Following the lines you'll see that these two inputs end up in an “odd” and “even” software register. This is then interleaved in the *spectrometer.py* script to form a complete spectrum. Data is output in normal frequency order, meaning that channel 0 (corresponding to DC) is output first, followed by channel 1, on up to channel $2^{N-1}-1$.

PARAMETERS

Size of FFT	How many points the FFT will have. The number of channels will be half this. We've selected 2^{10}=1024 points, so we'll have a 2^9=512 channel filter bank. This should match up with the pfb_fir block.
Input/output bitwidth	The number of bits in each real and imaginary sample as they are carried through the FFT. Each FFT stage will round numbers back down to this number of bits after performing a butterfly computation. This has to match what the pfb_fir is throwing out. The default is 18 so this shouldn't need to be changed.
Coefficient bitwidth	The amount of bits for each coefficient. 18 is default.
Number of simultaneous inputs	The number of parallel time samples which are presented to the FFT core each clock. We have 2^2=4 parallel data streams, so this should be set to 2.
Quantization Behavior	Specifies the rounding behavior used at the end of each butterfly computation to return to the number of bits specified above. Here we'll just use Round (unbiased: Even Values).
Overflow Behavior	Indicates the behavior of the FFT core when the value of a sample exceeds what can be expressed in the specified bit width. Here we're going to use Wrap as Saturate will not make overflow corruption better behaved.
Add Latency	Latency through adders in the FFT. Set this to 2..
Mult Latency	Latency through multipliers in the FFT. Set this to 3.
BRAM Latency	Latency through BRAM in the FFT. Set this to 2.
Convert	Latency through blocks used to reduce bit widths after twiddle and butterfly stages. Set this to 1.
Architecture	Set to Virtex5, the architecture of the FPGA on the ROACH.
Use less	This affects if complex multiplications use less multipliers or adders/logic. For the complex multipliers in the FFT, you can use 4 multipliers and 2 adders, or 3 multipliers and a bunch or adders. So you can trade-off DSP slices for logic or vice-versa. Set this to Multipliers.
Number of bits above which to store stage's coeff's in BRAM	Determines the threshold at which the twiddle coefficients in a stage are stored in BRAM. Below this threshold distributed RAM is used. By changing this, you can bias your design to use more BRAM or more logic. We're going to set this to 8.
Number of bits above which to store stage's delay's in BRAM	Determines the threshold at which the twiddle coefficients in a stage are stored in BRAM. Below this threshold distributed RAM is used. Set this to 9.
Multiplier Specification	Determines how multipliers are implemented in the twiddle function at each stage. Using `behavioral HDL` allows adders following the multiplier to be folded into the DSP48Es in Virtex5 architectures. Other options choose multiplier cores which allows quicker compile time. If selected, you can enter an array of values allowing exact specification of how multipliers are implemented at each stage. Leave this unchecked.
Use DSP48's for adders	The butterfly operation at each stage consists of two adders and two subtracters that can be implemented using DSP48 units instead of logic. Leave this unchecked.

REAL AND IMAGINARY BRAMS

no documentation online



The final blocks, **re_channel_bram** and **im_channel_bram** are shared BRAMs, which we will read out the values of using the *gpu_spec_init.py* script.

PARAMETERS

Parameter	Description
Output data type	Unsigned
Address width	2^(Address width) is the number of 32 bit words of the implemented BRAM. There is no theoretical maximum for the Virtex 5, but there will be significant timing issues at bitwidths of 13. QDR or DRAM can be used for larger address spaces. Set this value to 12 for our design.
Data binary point	The binary point should be set to zero. The data going to the processor will be converted to a value with this binary point and the output data type.
Initial values	This is a test vector for simulation only. We can leave it as is.
Sample rate	Set this to 1.

INPUTS/OUTPUTS

Port	Description
Addr	Address to be written to with the value of data_in, on that clock, if write enable is high.
data_in	The data input
We	Write enable port
data_out	Writing the data to a register. This is simply terminated in the design, as the data has finally reached its final form and destination.

CONTROL REGISTERS

no documentation online

There are a few control registers, led blinkers and snap block dotted around the design too:

1. **channel_select:** Select which channel to record (should be set between 0 and 511)
2. **start_capture** Toggle from 1 back to 0 to start a new data capture.
3. **capture_done:** Set to 0 by the hardware when a capture is still in progress and 1 when the capture is completed.

Hardware configuration

The tutorial comes with a pre-compiled bof file, which is generated from the model you just went through:

```
'gpu_spec_2010_Aug_11_1615.bof'
```

Copy this over to you ROACH *boffiles* directory, chmod it to *a+x* as in the other tutorials, then load up your ROACH. You don't need to telnet in to the ROACH, all communication and configuration will be done by the python control script.

The tutorial comes with a python file called *gpu_spec_init.py*. To use this, you need to have installed a few python libraries. If you haven't already, go through the instructions on

http://casper.berkeley.edu/wiki/Corr

I'd recommend installing **all** the packages, and documenting any trouble you have on the discussion page of the wiki. Also, iPython is used later on in this tutorial, so install that too.

Next, you need to set up your ROACH. Switch it on, making sure that:

- You have your ADC in *ZDOK0*, which is the one nearest to the power supply.
- You have your clock source connected to *clk_i* on the ADC, which is the second on the right. It should be generating an 800MHz sine wave with 0dBm power.

If set up correctly, it should look like the photo on the right.

gpu_spec_init.py

Once you've got that done, it's time to run the script. First, check that you've connected the ADC to *ZDOK0,* and that the clock source is connected to *clk_i* of the ADC.

Now, if you're in linux, browse to where the spectrometer.py file is in a terminal and at the prompt type

```
python gpu_spec_init.py <roach IP or hostname>
```

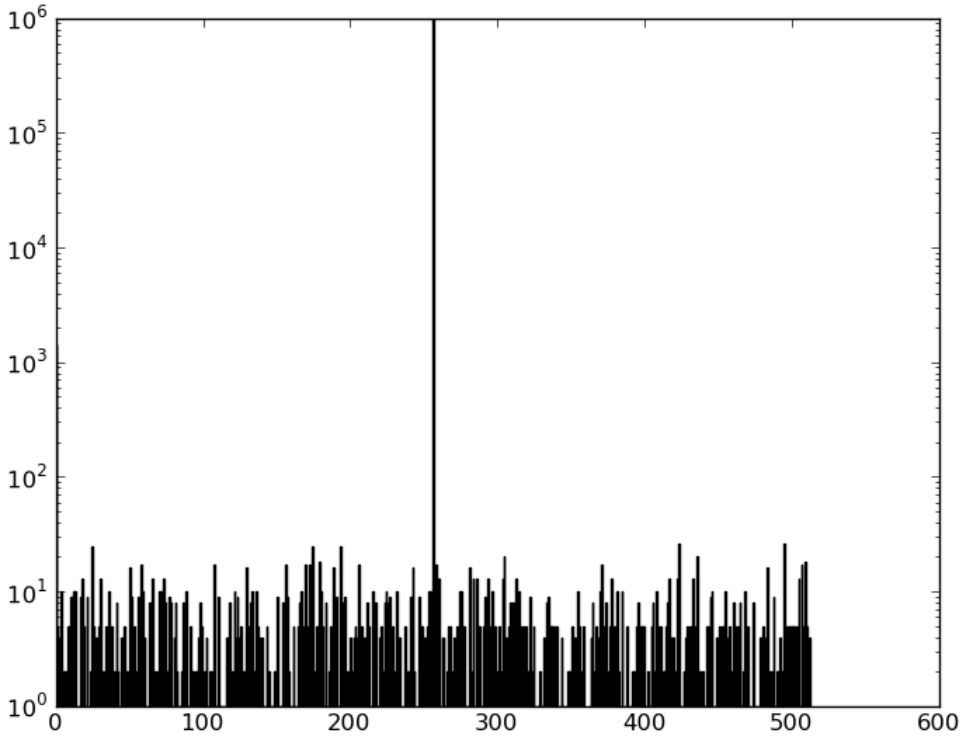
replacing *'<roach IP or hostname>'* with the IP address of your ROACH, or the serial if you've set this up. Refer to tutorial 3 for a detailed overview of ipython and katcp.

The script can be run with 3 options. -c CHANNEL_SELECT will allow you to set which channel the script will record by default. The data will be recorded in the data directory in a file named channel<selected_channel_number>_out. -p will open up a continuously updating pylab plot of the entire power spectrum to test the spectrometer. The script will use the precompiled boffile by default. To specify a different boffile use the option -f followed by the file name.

Running the script like this, with a 200MHz tone feeding the adc:

```
python gpu_spec_init.py -c 256 -p <roach IP or hostname>
```

will create a file named channel256_out in the data directory and will pull up a live updating plot of the complete power spectrum that looks like this:



The each line of the recorded data file contains continuous real and imaginary data from the same channel. The data folder has a sample recording named channel256_out that contains data recorded with a 200MHz tone feeding an iADC.

PART TWO

GPU Spectrometer

CUFFT

Now we will use the GPU to finely channelize this data. gpu_fft.c uses the cuda fft library to do a 2048 point 1 dimensional fft on our recorded data. Refer to the CUFFT library documentation for more information on the functions cuda has to offer.

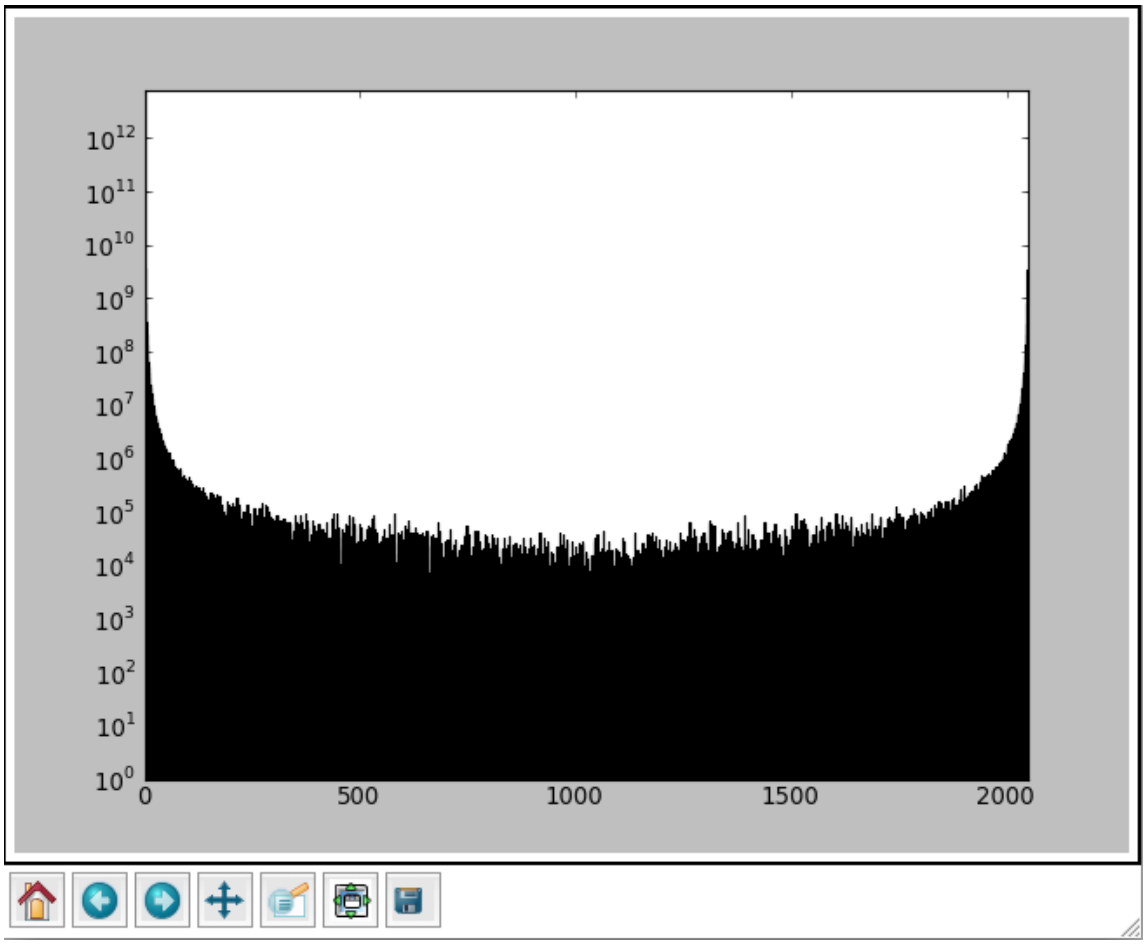
Here is an overview of what gpu_fft.c does (you should be able to refer to the comments in the file to follow along):

- Read in which channel to process from the command line
- Open the recorded data file data/channel<selected_channel_number>_out
- Create pointers for cufftComplex data. This is a struct containing 2 floats (to store real and imaginary data from each point) that the CUFFT library uses for complex ffts.
- Allocate space on the cpu and gpu using cuda functions. The space on the cpu is allocated using cudaHostMalloc rather than the standard malloc because it speeds up memory transfers between that CPU memory to and from GPU.
- Read the data from the file into the memory we allocated on the CPU.
- Create an fft plan (if you are familiar with FFTW this works in a similar way). This tells CUDA we want to do a complex to complex fft of length *fftlen*. It also allows multiple ffts to be batched together but in this case we will only do 1 transform.
- Move the recorded channel data from the CPU to the GPU.
- Execute the FFT
- Copy the FFT result from the GPU back to the CPU
- Store the FFT result to a file named data/channel<selected_channel_number>_spectrum
- Deallocate our fft plan and malloced memory

The makefile included in the src directory will use nvcc and the cufft library (included with -lcufft) to compile gpu_fft.c into an executable named gpu_fft.

plot_gpu_spectrum.py

Running *gpu_fft -c 256* will read in data from the data/channel256_out file, channelize it, and record the spectrum in data/channel<selected_channel_number>_spectrum. We can plot the power spectrum using plot_gpu_spectrum.py to see the finely channelized data from channel 256:



Conclusion

After completing this tutorial you should be able to channelize data on a ROACH and zoom in on a single channel using a GPU. To continuously feed data into the GPU, you will need to packetize the data and send it to the server over 10GbE. Refer to Tutorial 2 for more information on using 10GbE on the ROACH.