EN.605.417.FA17

Michael DiBerardino

Project Report

Repository

https://github.com/mdibera1/EN605.417.FA/tree/master/tuneFilterDecimate

Introduction

A method for performing a common digital signal processing (DSP) tasks using a graphics processing unit (GPU) was explored. When performing passive radio frequency (RF) data acquisition, there are typically three steps involved: tune, filter, and decimate. These processes must happen in real time so that an accurate representation of the RF energy is captured. Typically this is done using a Field Programmable Gate Array (FPGA) or a very high end real time processor. While being well suited for the task, FPGAs are difficult to design and are typically loaded with purpose built firmware that will have limited reuse. This is a good application for a GPU that can handle the parallel and real time requirements while still being software oriented.

Discussion

The first step involved in RF processing is to tune to the desired frequency. This is accomplished by using a complex multiplication to multiply the signal by a complex rotating phaser. The frequency shift property is seen in Equation 1, where X(f) is the Fourier Transform of x(t). Imaginary numbers cannot be used naturally in hardware, so Euler’s identity, seen in Equation 2, is used as two separate streams of data. This concept is used to move the signal of interest to 0Hz, which is the center of the processing window

$$\text{Equation1: } x(t)*e^{-2\pi f0t} <=> X(f+f_0)$$

$$\text{Equation 2: } e^{j2\pi ft}= COS(2\pi ft) + j*SIN(2\pi ft).$$

Decimation is the process of reducing the sample rate by low pass filtering and down-sampling.

Low pass filters restrict high frequency content in the signal while allowing low frequency content to pass through with little to no attenuation, depending on the complexity of the filter. In Figure 1, the frequency response of the 64tap Finite Impulse Response (FIR) low pass filter is shown. This filter was designed to allow the lower half of signals through and to remove the upper half of signals. Note that the spectrum is complex and that 0Hz is at the center of the plot. The x-axis has been normalized so that it has a domain of -1 to 1, corresponding to -Fs/2 to Fs/2.

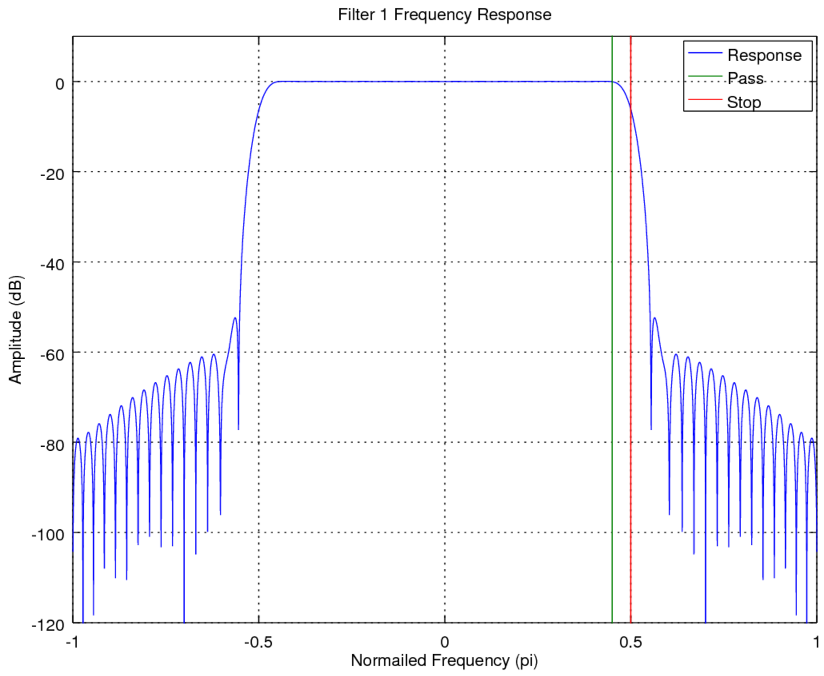

Figure 1

Down-sampling is the process of removing samples from the data. For example, down-sampling by two is the same as removing every second sample from an array, reducing the size of the array by half. This works as long as there are no signals that have a frequency higher than Fs/decimation_rate/2, or the new Fs/2. This is due to the Shannon-Nyquist sampling theorem[1], which states that in order to represent a signal with a frequency f, the sampling rate must be greater than of equal to 2*f. Since down-sampling reduces the sample rate by n, then the highest frequency that can be represented also is reduced by a factor of n.

Figure 2 shows the original signal which consists of 5 peaks. The two on the left side are at -100 and -200 kHz, and the three on the right are at 340, 350, and 360 kHz. The sampling rate has been set to 1MHz. After down-sampling, the three peaks on the right are greater than 250 kHz or Fs/4 and thus alias back into the negative side of the spectrum. As seen in Figure 3, the two signals at -100 and -200 kHz have not moved, but the other three signals have ended up in between. This is highly undesirable behavior and can lead to processing signals other than the one intended. The properly decimated signal is shown in Figure 4 and only consists of the two peaks.

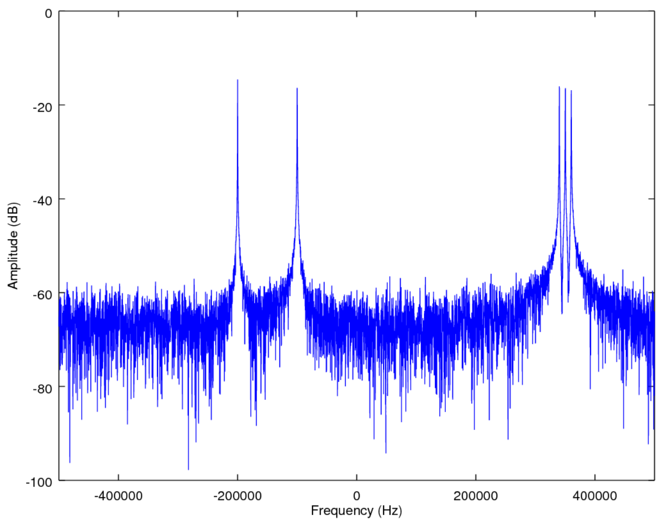

Figure 2

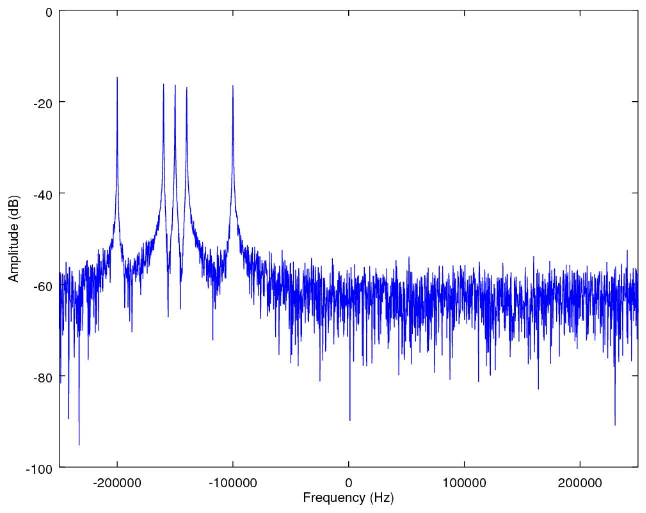

Figure 3

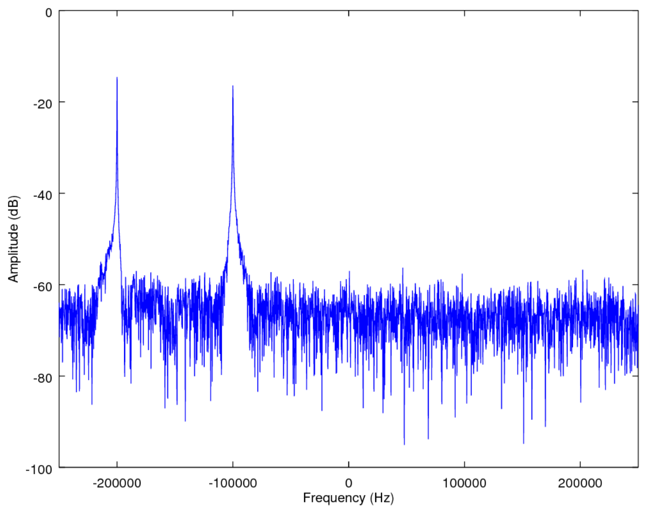

Figure 4

Approach

A tune, filter, and down-sample process will be performed in the GPU. Using the signal in Figure 2 as the starting point, the signal will be shifted to the left by 350 kHz. This will produce a spectrum with the three peaks centered around 0 Hz (at -10 kHz, 0 kHz, and +10 kHz) and the two peaks at -450 kHz and 450 kHz, placing them both out of band after decimating by 2. In order for success to be declared, there should only be three peaks around 0Hz. The +/-450 kHz would alias to +/-50 kHz if the low pass filter was not implemented correctly.

The Cuda code will be laid out in a way that promotes massive parallelism. A data size of 4096 samples will be used along with the 64 tap filter. This translates to 4096 blocks of 64 threads each. Each thread will implement the convolution as a single multiplication and copy from global to shared memory. The 64 threads in each block will then sum in parallel to result in 4096 data samples. This method of parallel summation, laid out by Mark Harris at Nvidia[2], sums n numbers in log2(n) time. The down-sample process can be implemented as a modulo of the block id number and a copy from shared to global memory.

A timing analysis will also be ran. Octave's complex multiply and decimate functions will be used to implement the same process as above except on the CPU. This will create a timing benchmark to compare the GPU times with. It will also create a spectrum that will be plotted alongside the GPU spectrum.

Results

As seen in Figures 5 and 6 below, the CPU and GPU resulting signals are nearly identical. There are only very minor variations in the noise floor. Only three peaks appear, which indicates that the two signals, originally at -100 kHz and -200 kHz, were properly attenuated.

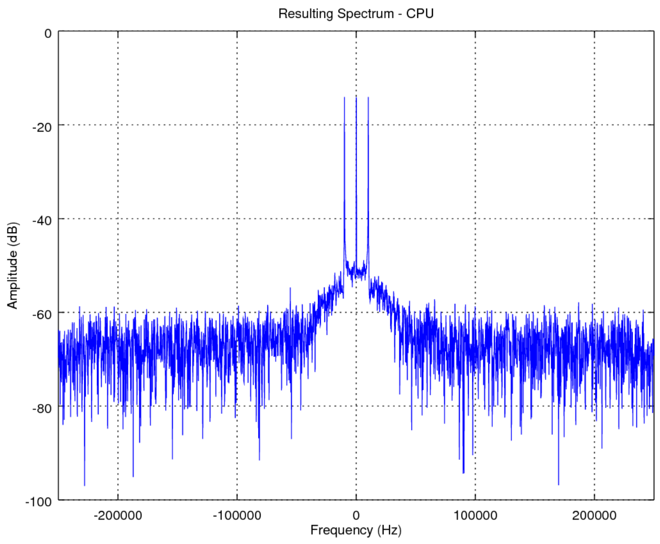

Figure 5: CPU Octave result

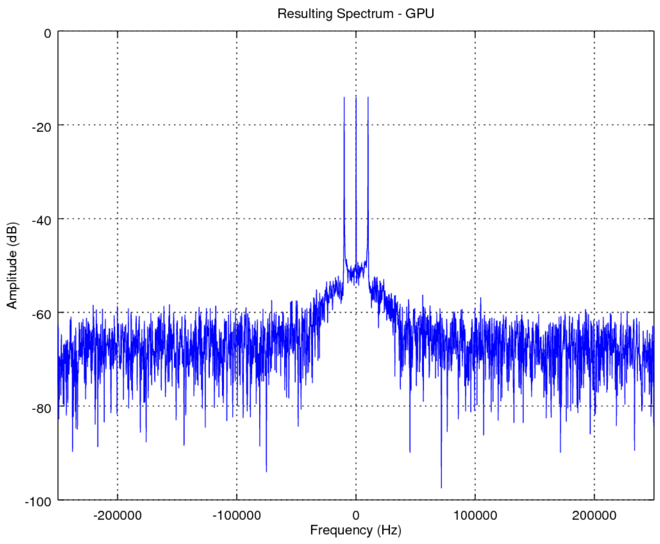

Figure 6: GPU Cuda result

Figure 7 shows the performance plot. The GPU has significantly shorter execution times due to parallelism. Note that both executions include time to generate and multiply by the frequency shift array to implement the frequency shift. The GPU loop includes the time spent copying data to and from the GPU, and the CPU includes time to allocate the frequency shift and data arrays.

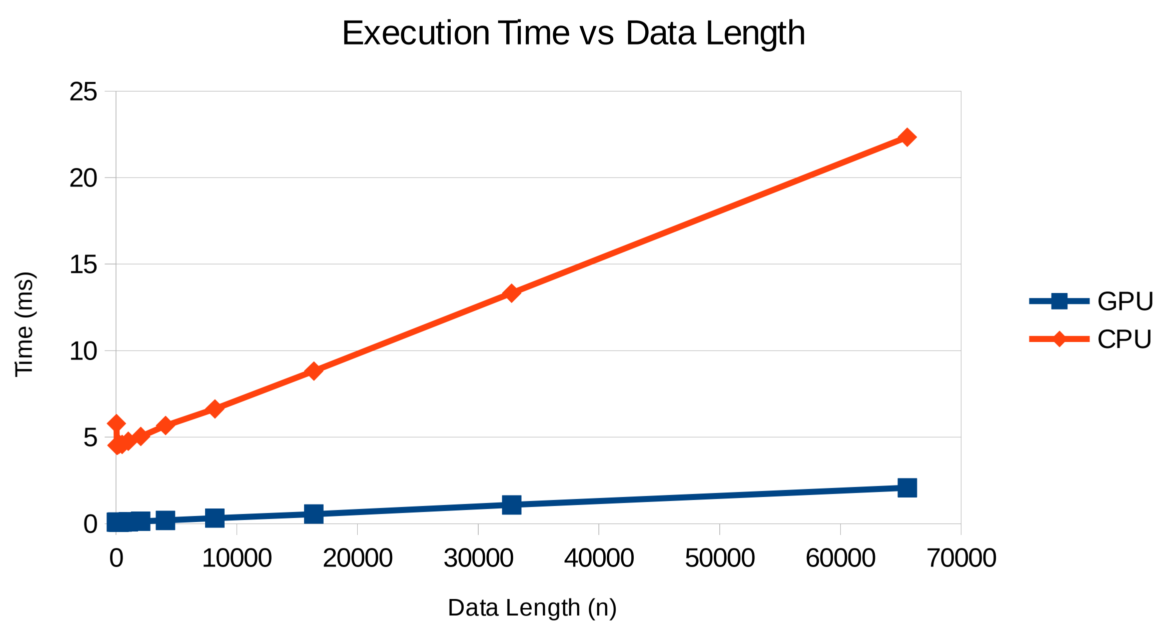

Figure 7

Conclusions

As seen in Figures 5 and 6, the CPU code was a success and looks identical to the data generated by Octave. But as seen in Figure 7, the GPU executes in significantly less time. This shows that the parallel implementation of the frequency shift array generation, single multiply per thread convolution, and parallel sum are efficient and accurate.

Even though white noise was used to generate the data, the same data file was used in both the CPU and GPU code. This eliminates changes from run to run in the noise floor. The two resulting signals are very similar, but are not exactly the same as seen in the minute differences in the noise floor. This could be due to roundoff error due to Cuda using a float type and Octave using a double type. Both algorithms are sourced from the same float type file.

References

[1] Wescott, T. (2016, June 20). Sampling: What Nyquist Didn’t Say, and What to Do About It. Retrieved December 10, 2017, from http://www.wescottdesign.com/articles/Sampling/sampling.pdf

[2] Harris, M. (2007, November 20). Optimizing Parallel Reduction in CUDA. Retrieved October 11, 2017, from http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf