Adaptive GPU-Accelerated Software Beacon Processing for Geospace Sensing

John Grasel – Harvey Mudd College

MIT Haystack REU Summer 2010

Abstract

Radio beacons on satellites can be used in conjunction with ground receivers to study the ionosphere. The flexibility of new wideband tuners and digital receiver platforms requires a modular, adaptable software chain to optimally process and interpret beacon overflight data. A python-based system was developed to track the beacon, filter noise, and convert the signal to baseband. The slow but intrinsically parallel nature of the process led to large performance gains when methods were ported to the Graphical Processing Unit (GPU) using a python wrapper of NVIDIA's CUDA programming language. This paper will discuss methodologies to port algorithms to GPU execution as well as show results for representative beacon overflights in the Westford, MA vicinity.

Background

The large-scale structure ionosphere consists of layers of ions and electrons, but within these are small-scale regions of irregular electron density. As Figure 1 on the next page demonstrates, these density structures cause RF (radio frequency) signals propagating through them to experience random changes in amplitude and changes in phase, called scintillations. Atmospheric scientists use satellite beacon signals to study scintillation, the amplitude and phase variation of a radio signal imposed by ionospheric variations. Since scintillation causes a loss of signal integrity, it is important to understand its effects. Scintillation directly affects the Global Positioning System (GPS) and the High Frequency (HF) communication band signals which are to communicate with aircrafts flying over the poles and the space shuttle during landings. In addition, electron density information can be extracted from scintillation data. Profiles of electron density are a key component of space weather, a broad phenomenon that studies the effects of solar conditions on our atmosphere, including auroras and geomagnetic storms.

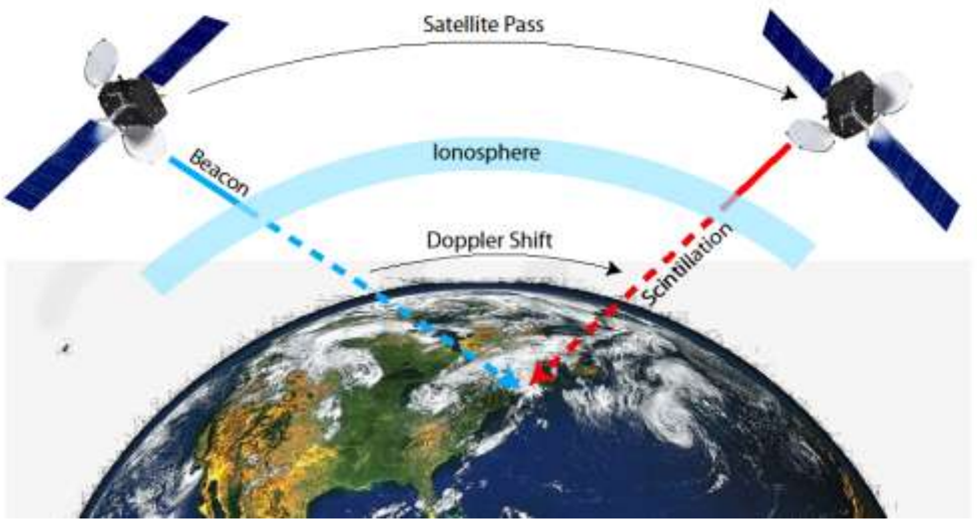

Figure 1 - A Satellite Pass and Ensuing Doppler Shift and Scintillation

The recording and processing of beacon data is done with a beacon chain because of its linked but modular nature. While this project focuses on software signal processing, it is necessary to have a basic understanding of the hardware further up the chain. The satellites OSCAR 25, DMSP F15, and RADCAL all have RF beacons at 150 and 400 MHz shown in the figure above. As the beacon passes overhead, its frequency is Doppler-shifted and converted into electrical current by an antenna. The signal is mixed to an intermediate frequency (IF) and filtered around the beacon frequency. The analog to digital converter (ADC) samples the analog signal, and the digital receiver downsamples the signal to a lower frequency, typically 100 kHz. The data is fed into an Ethernet backend, where a network computer begins the software signal processing. The downsampling is necessary to handle the amount of data being sampled – while there are ADCs that can directly sample RF beacons, the rest of the chain would have to handle 1.6 GB/s of information, which increases the amount of hardware required and software runtime. Since the satellite beacons have a typical bandwidth of 20 kHz, they fit into a 100 kHz band.

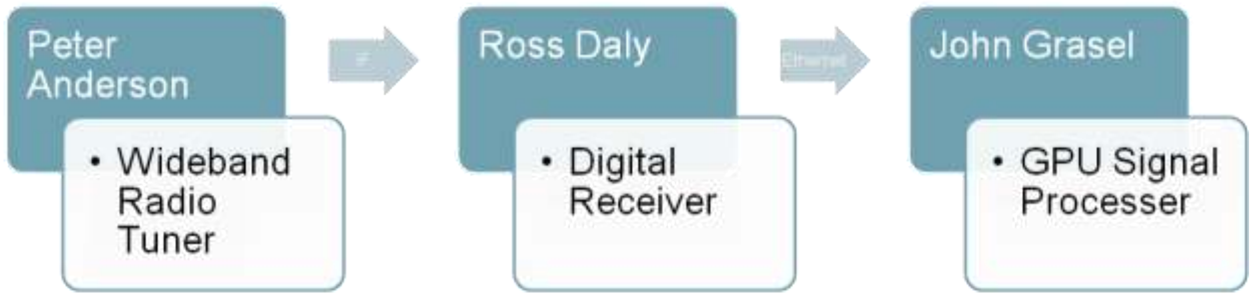

Figure 2 - The Beacon Chain

The traditional process for studying scintillation has been through dedicated hardware, but faster computers and powerful scientific libraries have allowed more work to be shifted to software written in high-level languages like Python. In addition to decreasing development time, the use of software allows for the data processing to be highly configurable. Software has much greater scalability as well; not only can it be cloned and shared freely among different users, but advances in computer hardware will increase software's performance with little or no work to the programmer. Finally, in software, data can be reprocessed or processed with using different configurations.

The primary goal of this project was the construction of a software signal processer for satellite beacons. The secondary goal was to accelerate the code using the Graphical Processing Unit, or GPU.

Design

The code production started with a high-level block diagram and continued with functions implementing each block. The pure Python block diagram is shown in the left half of Figure 3 on the next page. Too much data is collected from a satellite pass to process it all at once, so a Beacon Block contains a portion of the data. As such, it is the central object on which work is performed through the chain. The voltage data is stored in a two-dimensional array whose rows contain voltage data. The width of the Beacon Block, along with the sample rate, thus determines frequency and time resolution. The implementation of the Beacon Block is convoluted because the single class must operate on both the GPU and CPU. Since these classes contain different variables, the Beacon Block's methods become obfuscated with conditionals. If I were to rewrite the class, I would write an interface Beacon Block and populate it with code common to both the GPU and CPU, like the loading of meta data. This interface would be implemented in separate classes for the GPU and CPU code. This would ensure that the two were interchangeable and allow for conversion between them.

The instantaneous bandwidth of the signal is a measure of the width of frequencies in a signal. It can be used as an indicator of when the beacon is no longer being tracked. Since this function relies on no other variables other than voltage and its output is not used elsewhere in the program, it is a candidate for being executed in a parallel separate thread on the CPU or run on the GPU.

Since so many of the blocks utilize the Fast Fourier Transform (FFT), it's a very important component. Scipy's and Numpy's FFT libraries are used, and an fftshift function a la MATLAB was written in Python.

The beacon tracking process is divided into two main parts: tracking the satellite beacon over time and down converting the signal. The satellite beacon is tracked in the frequency domain, so an FFT is applied to each array in the Beacon Block, shown in Figure 5. The expected frequency of the satellite over time is calculated, and a Gaussian window is applied to each array in the Beacon Block around the expected frequency, depicted in Figure 6. This process filters noise sources before peak tracking. The software package PyEphem is used to calculate the satellite positions, and the frequencies are derived from this information. The satellite beacon is tracked via a method called Dynamic Programming (DP), shown in Figure 7. If the spectrogram of the signal is considered as a three-dimensional surface, the line tracking the peak should be the one that stays at the highest power for the longest time. DP allows for this line to be calculated in n-squared time as opposed to the naïve recursive method's exponential time, and it can utilize parameters such as the results of previous Beacon Blocks increase the accuracy of the tracking. In addition, the calculation yields a confidence indicator that accurately indicates the instantaneous strength of the tracked signal. This is a large advantage over closed-system beacon trackers that fail quietly. Finally, DP's implementation explicitly involves a continuity constraint that limits the space of possible beacon tracks to the set of those that are feasible. Unfortunately, the DP method is not inherently parallel since the peak at any one point depends on the peaks of previous points.

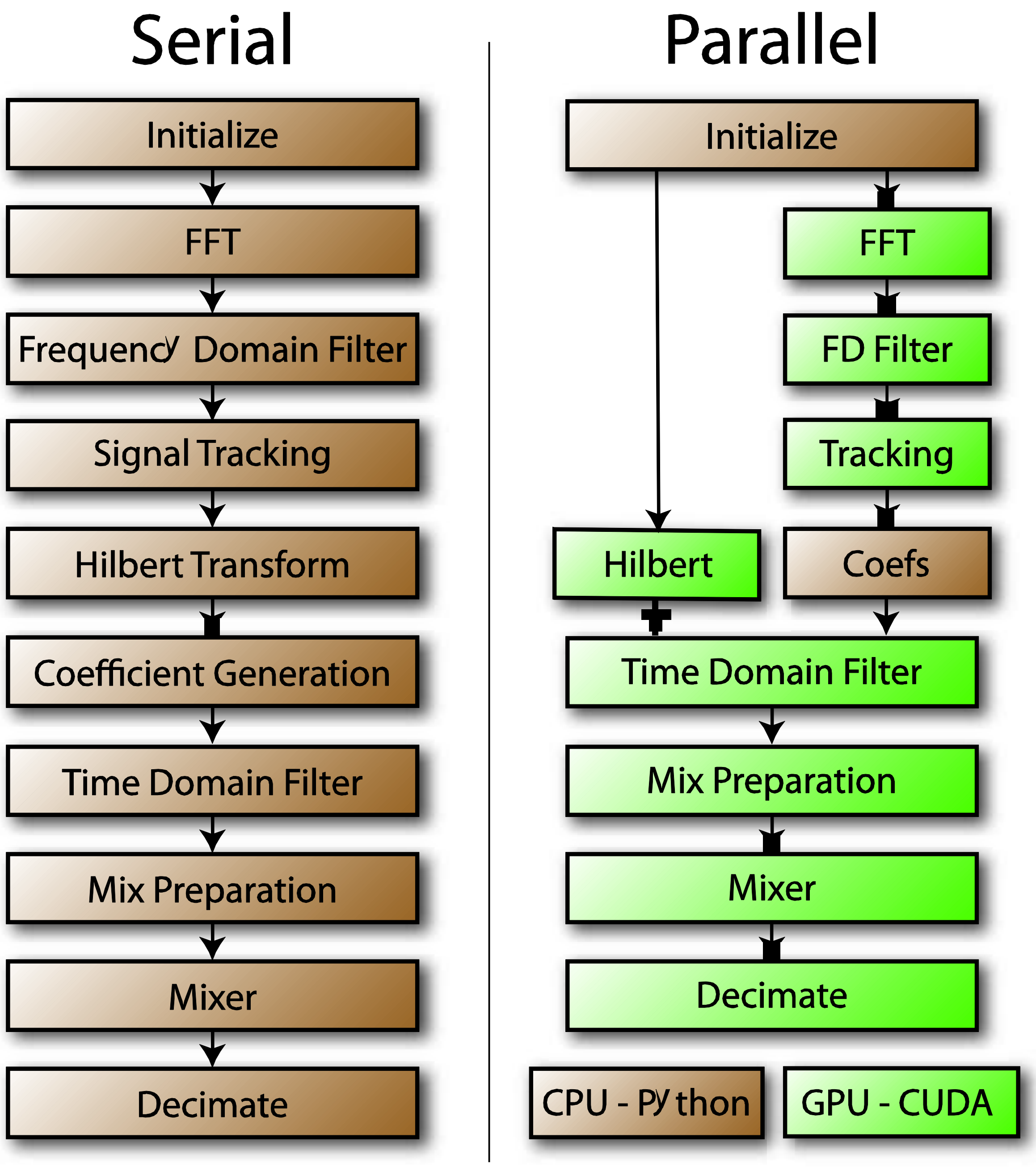

Figure 3 - The Program Model for Serial and Parallel Versions of the Processing Software

The second part of the process is the down conversion of the signal to baseband. First, a preselector filter removes signal components around the tracked peak in the time domain. This is implemented through a Finite Impulse Response (FIR) bandpass filter, shown in Figure 7. An FIR filter's output is a weighted sum of the current and a finite number (known as the order) of previous values of the input. The weights are generated using a SciPy package, though they correspond roughly to the inverse Fourier transform (IFFT) of the desired filter response. A higher order filter has sharper rolloff, but it requires more computation and has increased delay. Since each element of the output is a direct function of elements of the input, the FIR filter is well-suited for parallelization on the GPU.

The FIR filters generated by SciPy select both positive and negative frequencies, so a Hilbert Transform, shown in Figure 8, is necessary to remove the negative frequency components. This implementation performed the transform in the frequency domain – each array is thrown through a Fourier Transform, its negative frequency components killed, and then through an inverse Fourier Transform.

A real signal mixer works via the principle that the product of two signals with two frequencies is the superposition of signals with frequencies given by the sum and difference of the two input frequencies. Since we are trying to take a signal with known frequencies over time and mix it to baseband, mixing it with a cosine wave of the same frequency will yield a signal at baseband plus a signal with twice the known frequency. However, we don't want the double frequency tone, so we mix our Beacon Block arrays with a complex exponential of negative the beacon frequency. This effectively kills the Doppler component of the beacon and centers the signal on baseband. The application of the decimator to our Beacon Block is shown in Figure 10.

Finally, the signal is decimated through a process known as integer downsampling, shown in Figure 11. A signal decimated by a factor of M consists of one sample out of every M points. This reduces the signal's bandwidth by a factor of 1/M. We don't have to worry about Nyquist's Sampling Theorem's folding or aliasing because the combination of the FIR and the baseband signal mixer effectively function as a low pass filter.

Porting Python to the GPU

Unlike the CPU, which specializes in logic and program flow, the GPU is specialized for compute-intensive, highly parallel computation. CUDA is a C-subset designed to give the programmer access to the GPU's features in an easily-scalable manner. Functions are called kernels, and one kernel runs on the GPU at a time. A kernel runs on a two dimensional grid of blocks. Each block is a three dimensional container for threads. Threads within a block share local memory and can easily be synchronized. Blocks are required to be executable in any order. The kernel code is written for an arbitrary thread in an arbitrary block, and CUDA executes the kernel for each thread in each block in parallel.

PyCuda is a Python wrapper for the CUDA C library. It opens the entire CUDA library to Python bindings, and includes other convenient features. GPUArrays have all the functionality of Numpy arrays but can be passed to CUDA functions. For example, the arrays have their operators overloaded with CUDA-accelerated functions. In addition, PyCuda provides accelerated element-wise math for GPUArrays, like scalar and vector products and trig functions. PyCuda also provides a simple syntax for Elementwise Kernels and Reductions so the programmer can avoid writing CUDA C where possible. More complicated functions can be written in CUDA C as a Python string and compiled and run from within Python.

When writing GPU-accelerated applications, prototyping in Python makes writing and debugging the final product easier. In addition, any statement that can be expressed using Python's map, reduce, or list comprehensions can be written in PyCuda using Elementwise Kernels and Reductions. More complicated functions can be converted using a simple formula. Say you have a Python function that operates on a two-dimensional array:

```
def function(data):
    (y, x) = data.shape
    for j in xrange(y):
        for i in xrange(x):
            # operations on data[j][i]
```

and you want to write a version for the GPU. We start by defining an arbitrary vector block dimension. All this means is that the elements data[0][0], … , data[31][31] will be operated on in the same block.

```
threadBlockDim = (32, 32, 1)
```

Now we want to verify that our block dimension isn't greater than our array dimension, and if it is, we crop it down. Strictly speaking, this isn't essential, but it prevents lots of unnecessary threads from spawning.

```
if threadBlockDim[0] > x:
    threadBlockDim = (x, threadBlockDim[1], 1)
if threadBlockDim[1] > arraysPerBlock:
    threadBlockDim = (threadBlockDim[0], y, 1)
```

Now that we have the dimensions of a single block, we can calculate how many blocks we need in our grid.

```
gridx = int(math.ceil(x / threadBlockDim[0]))
gridy = int(math.ceil(y / threadBlockDim[1]))
```

The CUDA code is then generated in a Python string formatting. First we declare the function using the __global__ keyword, which specifies it is called externally from Python as opposed to internally from CUDA. The return type must always be void for __global__ functions, so any "returned" values must be modified inputs. Notice the %i statements allow for Python variables to set CUDA constants. Since this function is called once for each element of the array, a single Python variable lookup may often be faster than multiple CUDA lookups. This technique is known as metaprogramming.

```
code= '''
    __global__ void function(int[%i][%i])
    {
```

The next step calculates the index of the array that a specific thread is working on

.

```
        int i = blockIdx.x * %i + threadIdx.x;
        int j = blockIdx.y * %i + threadIdx.y;
```

Recall that the grid dimensions gridx and gridy span the array, because of the ceiling operation performed in their calculation, they likely include extra elements not in the array. To avoid mistakenly accessing array elements outside the array dimensions, we end the computation of any thread whose indices are outside of the array dimensions.

```
        if(j >= %i || i >= %i) return;
```

Finally, we perform whatever options we needed to perform, and use Python's string formatting to hardwire constants into the CUDA C code.

```
        # operations on data[j][i]
    }
    ''' % (y, x, threadBlockDim[0], threadBlockDim[1], y, x)
```

Code. Next, we compile the code, and then we can run it on GPUArray inputs:

```
code = pycuda.compiler.SourceModule(code)
gpu_function = code.get_function("function")
gpu_function(array,block=threadBlockDim,grid=(gridx, gridy))
```

Problems with CUDA

While CUDA and PyCuda are remarkably easy to use, there are several issues that must be considered before using them to accelerate an application. First of all, there must be enough parallel elements to make memory transfers from CPU to GPU worthwhile. Even though the PCI-X connection between the CPU and GPU is very fast, the latency of the connection slows down data transfer between the two. Add in the size of the data that is typically processed on the GPU and you have a huge bottleneck. In addition, practically no existing CUDA libraries except for CUFFT, a CUDA FFT library, and CUBLAS, a CUDA linear algebra library. Unfortunately, these elements are not compatible with PyCUDA. The problem exists because the CUDA Runtime API and the CUDA driver API are not compatible with each other. PyFFT is a CUDA-accelerated FFT library, but it currently only works for powers of two. Even if CUDA lends a huge performance boost for applications, it is often very arduous porting all of the external libraries used. Finally, when coding for the GPU, be sure to avoid code branching, which includes conditionals, at all costs. Up to 32 threads are executed simultaneously on each warp, a group of threads that share the same set of instructions. Essentially, instructions are simulcasted to all threads in the warp, so branching leads to idle threads.

GPU Results

Porting individual functions to the GPU resulted in modest speedups eroded by slow memory transfer between the CPU and GPU. The GPU-enabled processing is 70% faster than the CPU version, and it processes data at over 600,000 samples per second. The speedup for individual functions is shown in the figure below. The primary bottleneck in the GPU processing is the beacon tracking, which is currently slower than the CPU version because of the memory transfer involved. If the beacon tracking was ported to the GPU, the GPU performance would improve significantly.

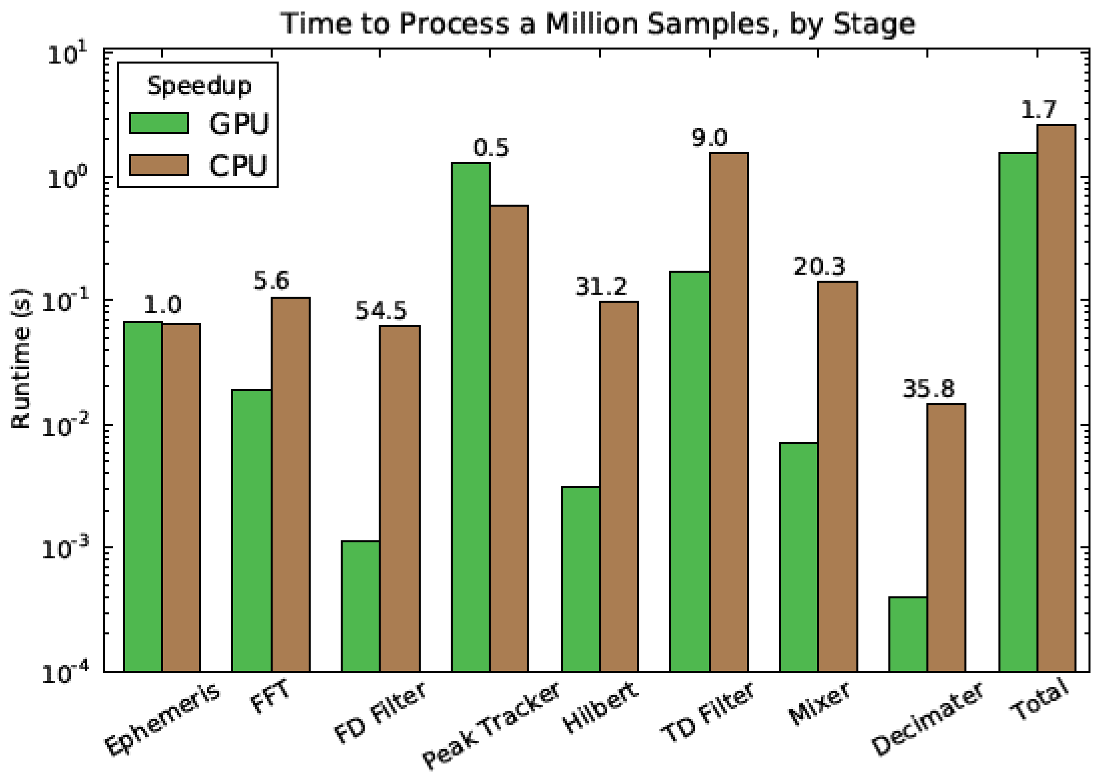

Figure 4 - Results of the Runtime of Serial and Parallel Versions of the Beacon Processing Software

Conclusions

A working configurable software radio processor was developed in Python and accelerated in PyCuda. The peak tracking with Dynamic Programming proved very effective, and the use of PyCuda sped up the processing by 70%, a rate 6 greater than necessary to perform real-time signal processing. CUDA is simple and powerful enough language to become mainstream in data-intensive applications, and PyCuda is a very effective way to access it.

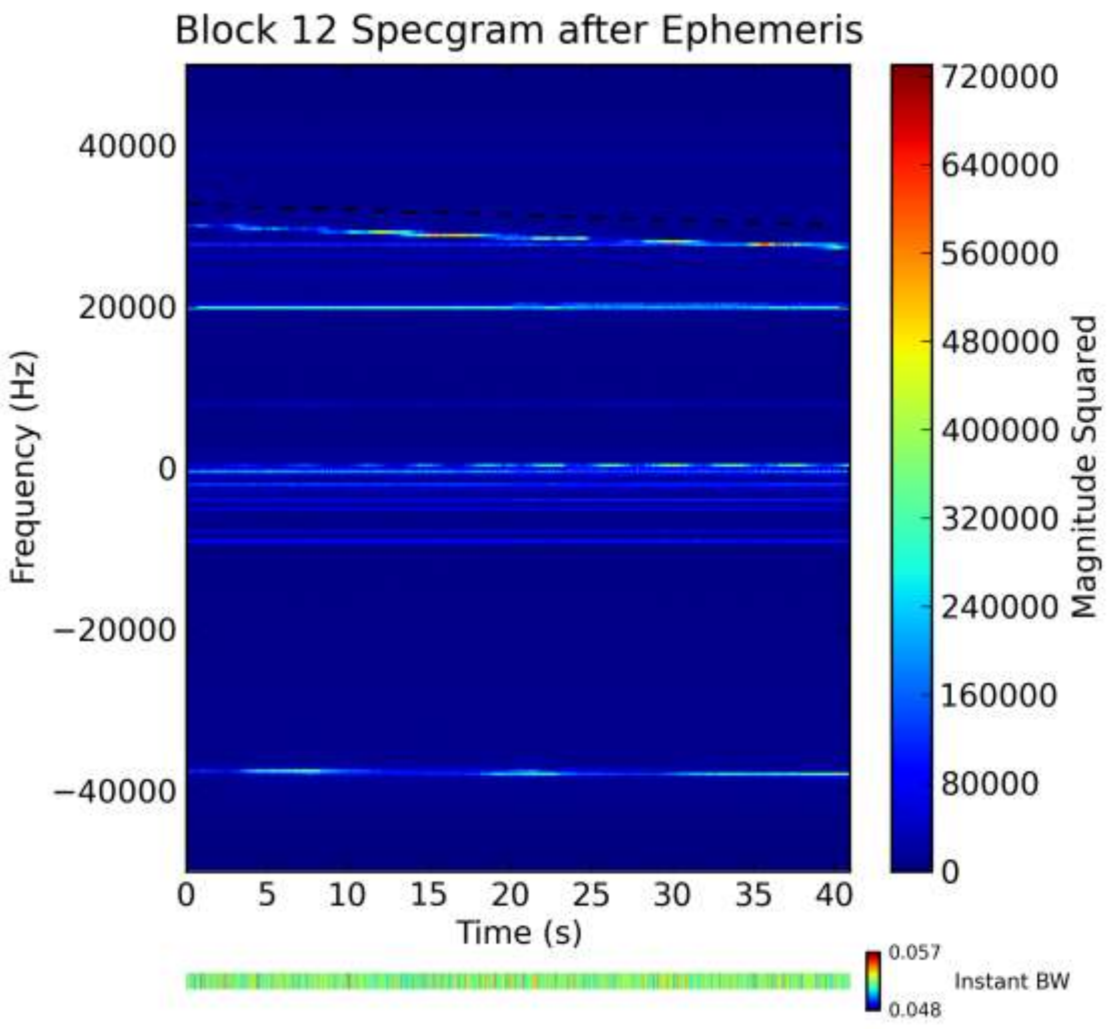

Figure 5 - Input Signal Spectogram. The dashed black line is generated from ephemeris and represents where the satellite beacon is predicted to be.

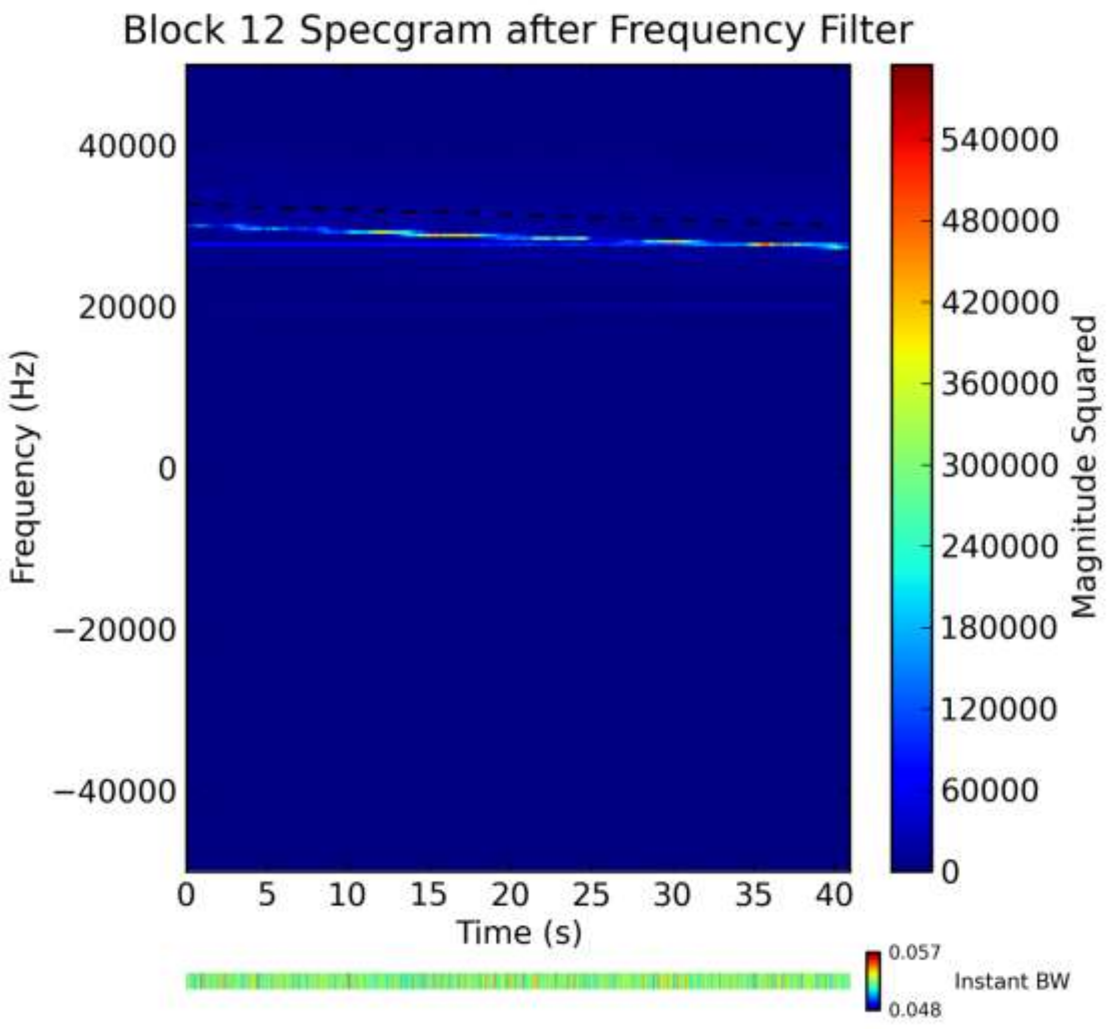

Figure 6 - Applying the Frequency Filter. Applying the filter around the Ephemeris data removes unwanted tones that may be stronger than the signal.

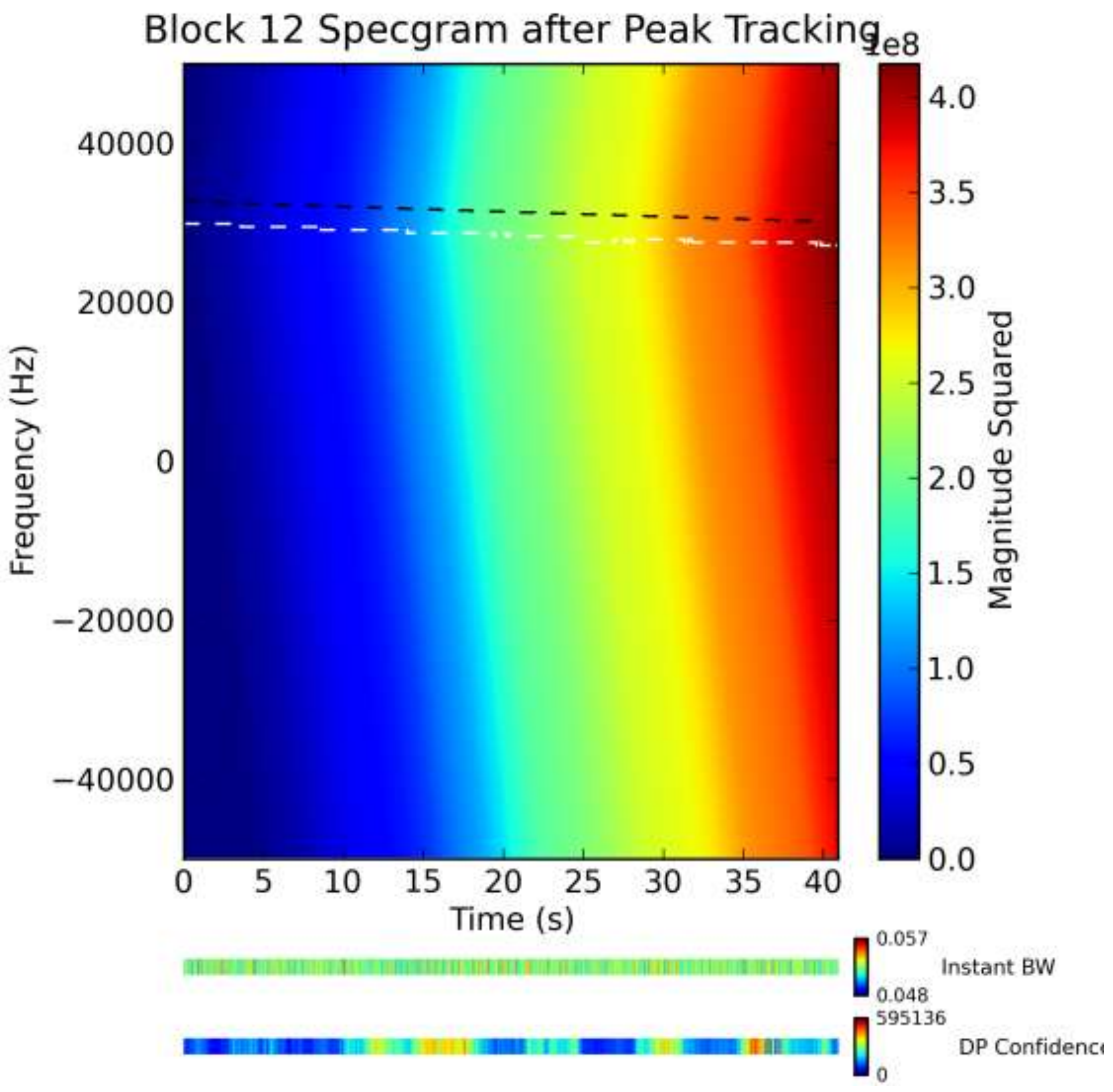

Figure 7 - Tracking the Beacon. The white dashed line represents the tracked tones, and the DP Confidence plot represents the strength of the tracked signal over time.

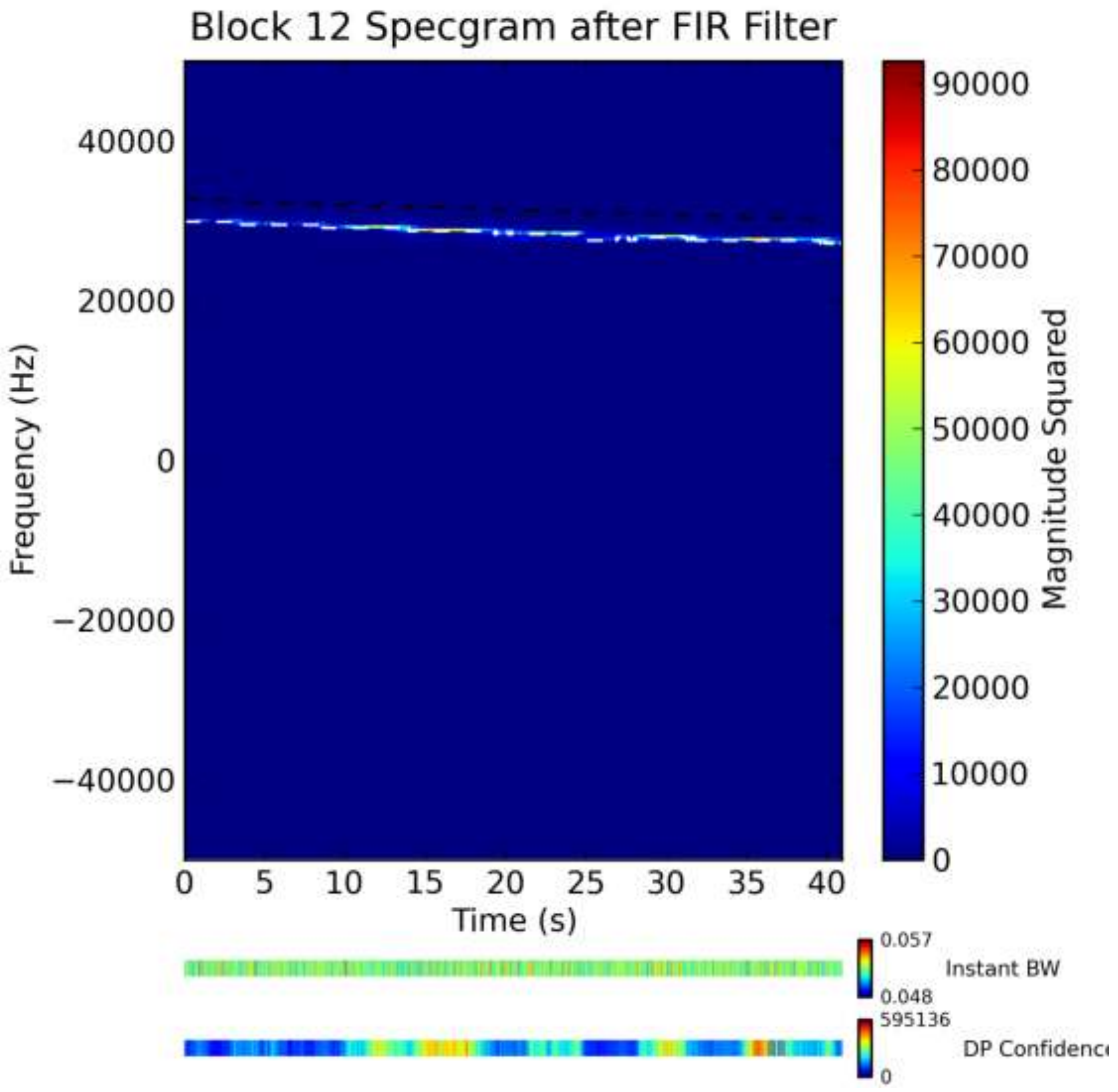

Figure 8 - Applying the Hilbert Transform. The Hilbert Transform removes negative frequency components from the signal, shown by the empty bottom half of the spectrogram.

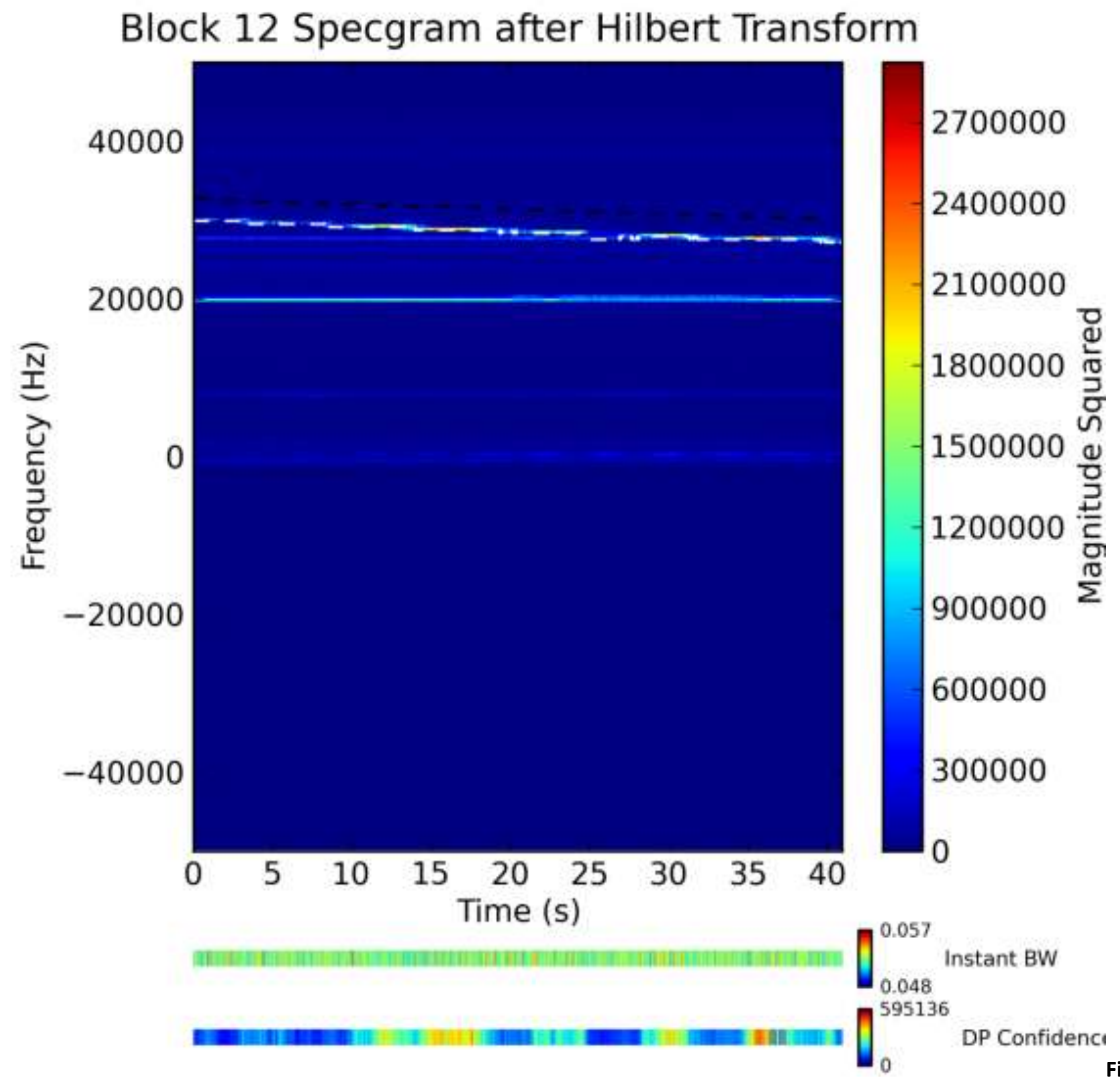

Figure 9

Figure 9 - Applying the FIR Filter. Filtering the data around the tracked signal reduces noise and will reduce aliasing artifacts in the decimation process.

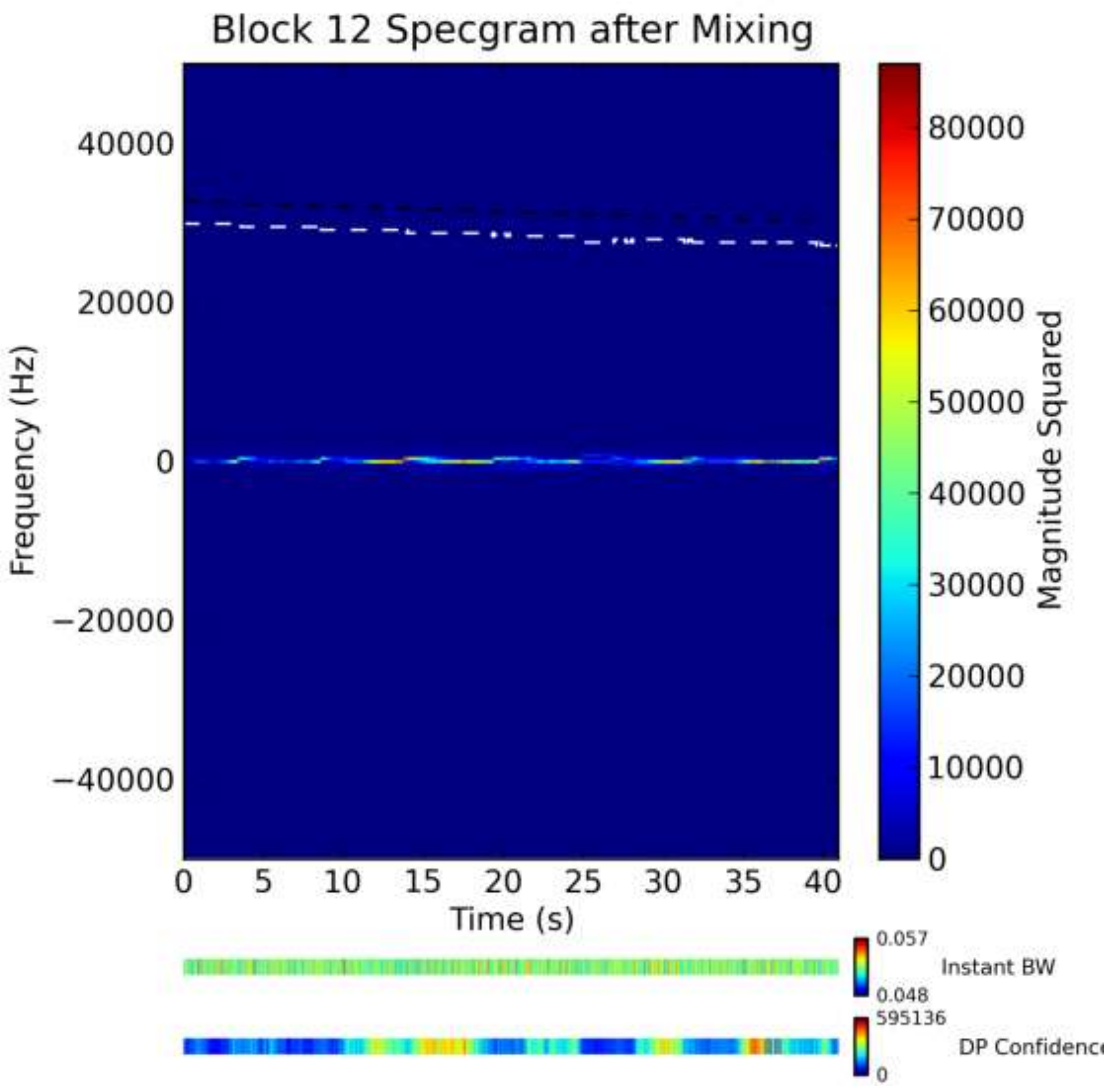

Figure 10 - Applying the Mixer. The signal is mixed to baseband, or 0 Hz.

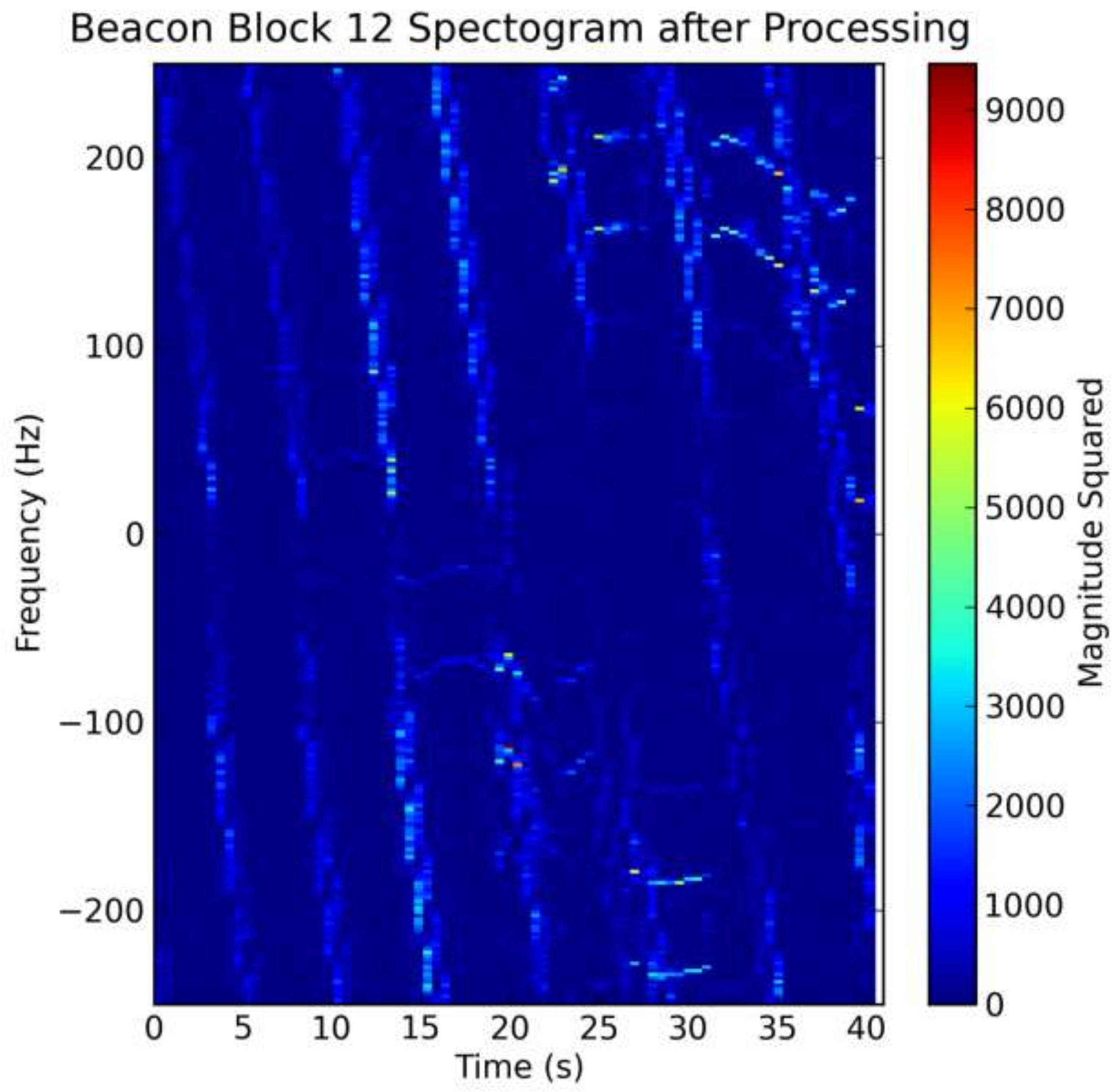

Figure 11 - Applying the Decimator. Decimating the signal reduces the signal bandwidth from 100 kHz to 500 Hz.