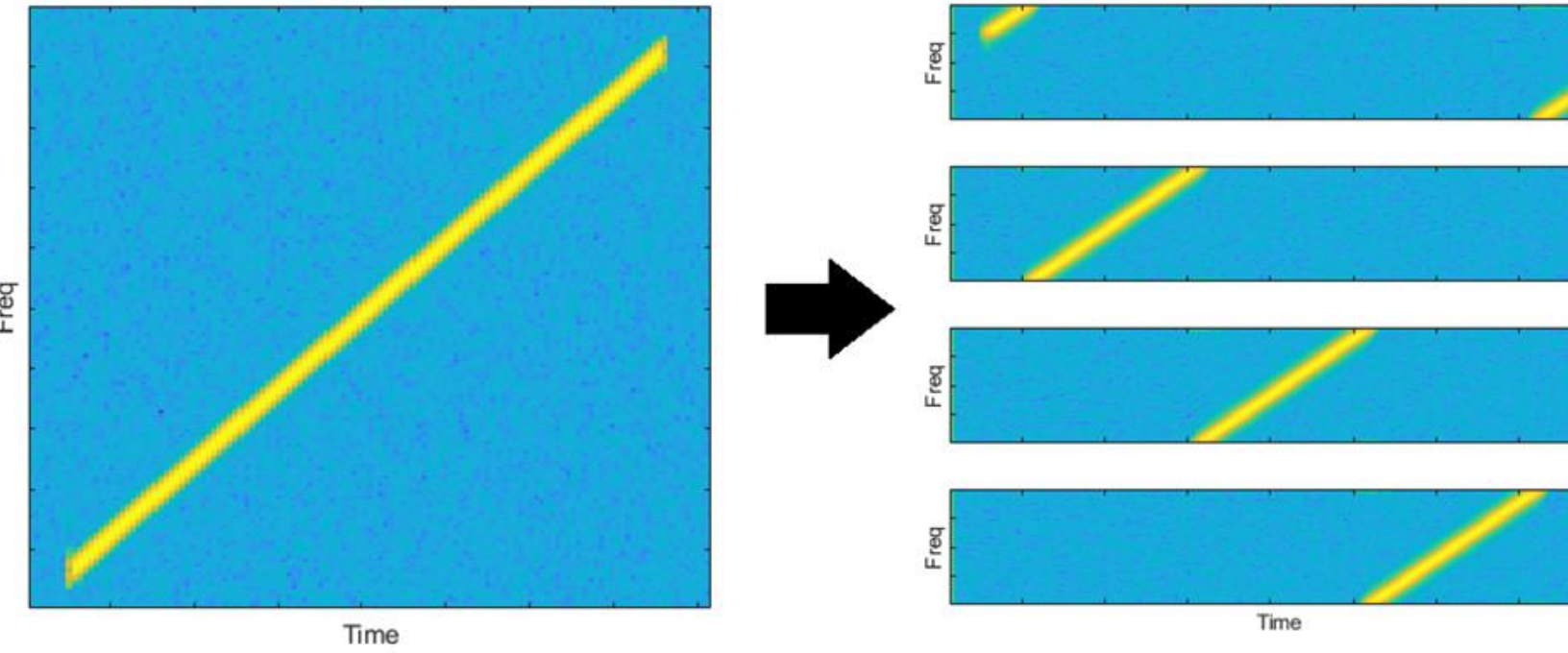
Signal Hound®



Signal Hound Channelizer
Manual

Signal Hound Channelizer Manual

Published 2/2/2021

1502 SE Commerce Ave, Suite 101
Battle Ground, WA
Phone 360-313-7997

1 Description

Signal Hound is releasing a real-time CPU-based RF baseband channelizer library for use with Signal Hound spectrum analyzers and receivers. The channelizer is a 1-to-M I/Q channelizer with arbitrary channel counts up to 2048 channels and configurable multi-threading. The channelizer can sustain input rates in the 100's of millions of input samples per second on standard desktop processors, making it ideal for real time monitoring and intelligence applications.

Libraries are available for use on Windows and Linux.

2 Theory

The channelizer converts 1 input I/Q data stream with sample rate ***S*** into ***M*** I/Q data streams each with sample rate ***S/M***. The input channel is divided into ***M*** equidistant and spaced channels. For example, a 50MS/s I/Q input could be channelized into 50 1MS/s output channels, or 2000 25kS/s output channels.

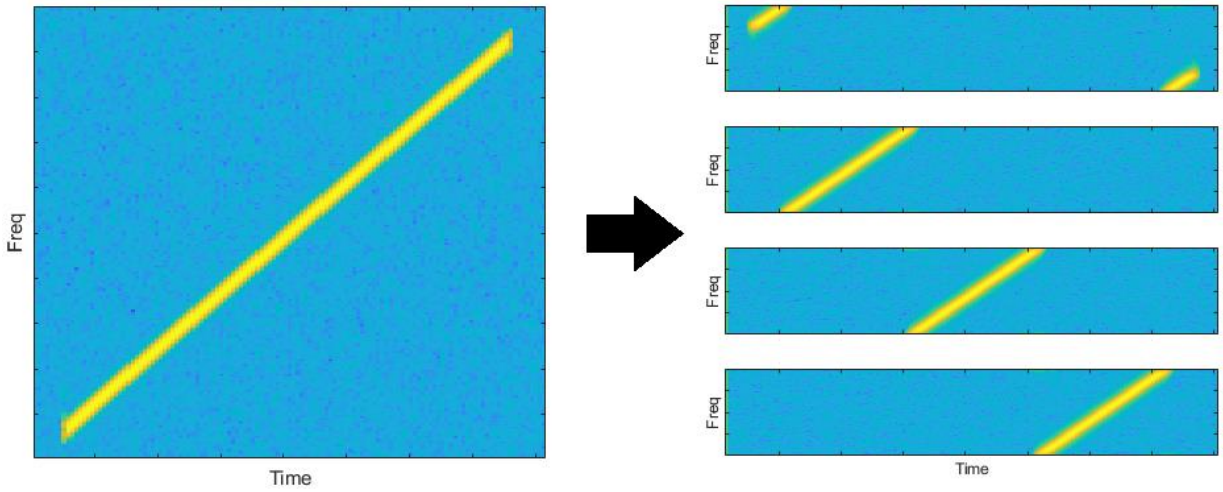


Figure 1: Input channel containing a frequency ramp is channelized into M=4 output channels.

3 Implementation

The channelizer uses a traditional polyphase filter bank channelizer approach. Multi-threading is accomplished by processing overlapping inputs in each thread in a standard thread pool implementation.

The output of the FFT is neither bit reversed nor are the halves swapped, which means the channels are provided linearly from negative to positive frequency. Rephrased, for a given channel count of M, an M-point FFT is used, and the channels are ordered,

`-M/2, -M/2+1, … -1, 0, 1, …, M/2 – 1, M/2.`

4 Performance Notes

- The channelizer is optimized for Intel processors. The channelizer may not run as well on AMD CPUs.
- The channelizer uses AVX intrinsic instructions. The CPU must support AVX1.
- Performance is affected linearly with filter size. Ideal filter sizes are small, 16 to 32 taps per channel. A larger filter will have a steeper roll-off near the channel boundaries at the expense of more processing.
- Performance is highly dependent on input size. The channelizer operates on fixed input sizes which are selectable by the user. As the input size approaches the per-thread cache size, performance decreases (see performance plot). Input sizes that are too small will experience overhead costs associated with calling into the API too often. See the usage section for more information on input size.
- Signal Hound has provided a benchmark function which can be used to benchmark various configurations to find the best one for a given system.

5 Performance

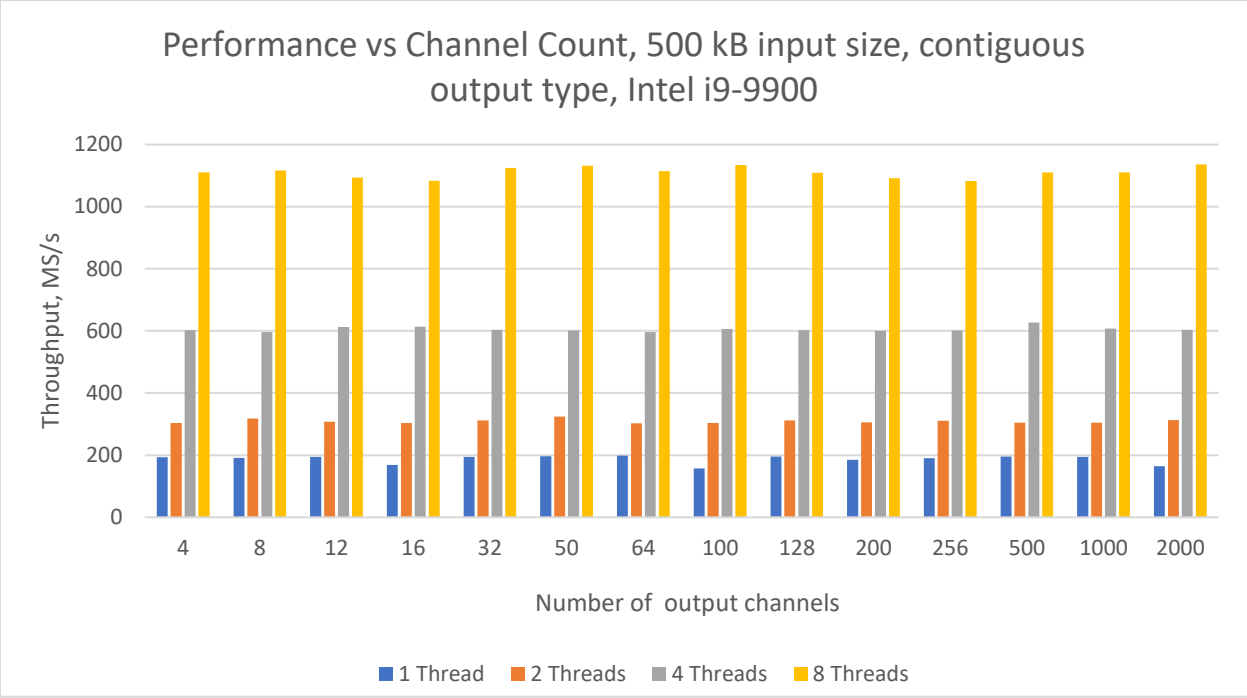


Figure 2: Performance remains relatively constant for different channel counts and performance increases close to linearly with the number of processing threads. Note that the channel counts used are easily factorizable numbers, using non-factorizable channel counts such as large prime numbers will affect performance negatively.

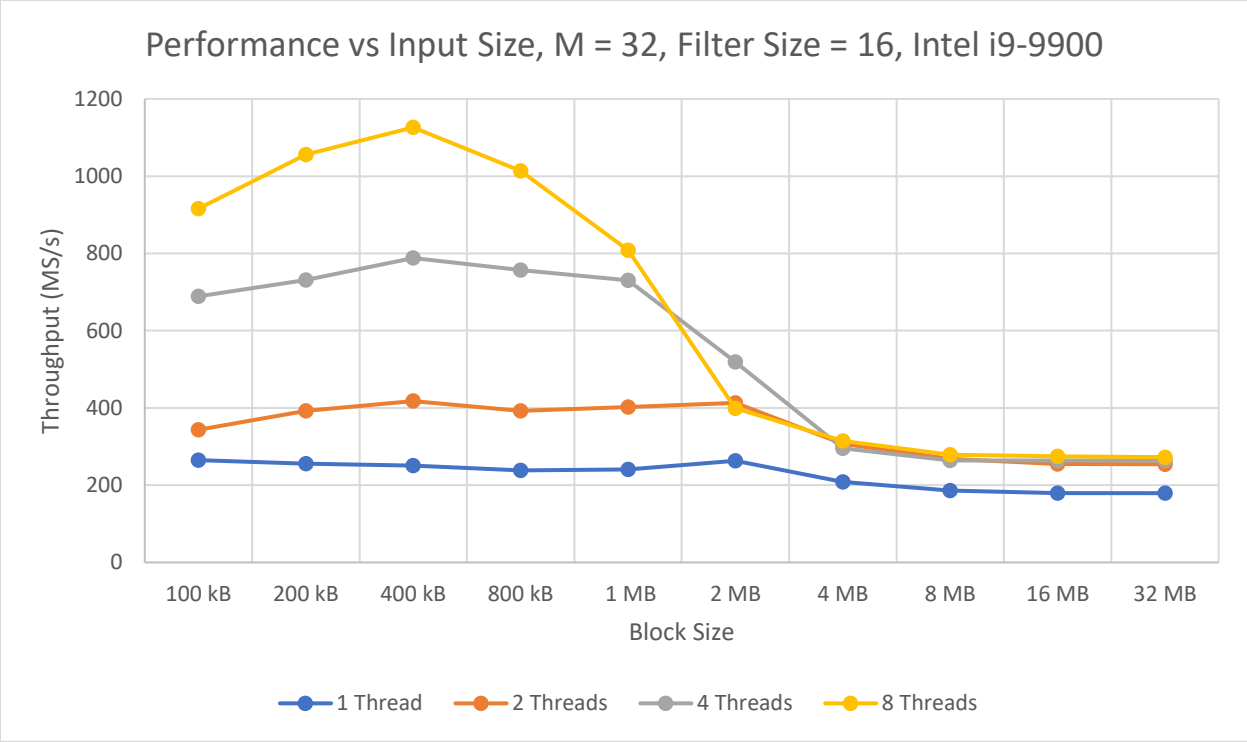


Figure 3: Input size plays an important role in performance. Several factors affect the ideal input size.

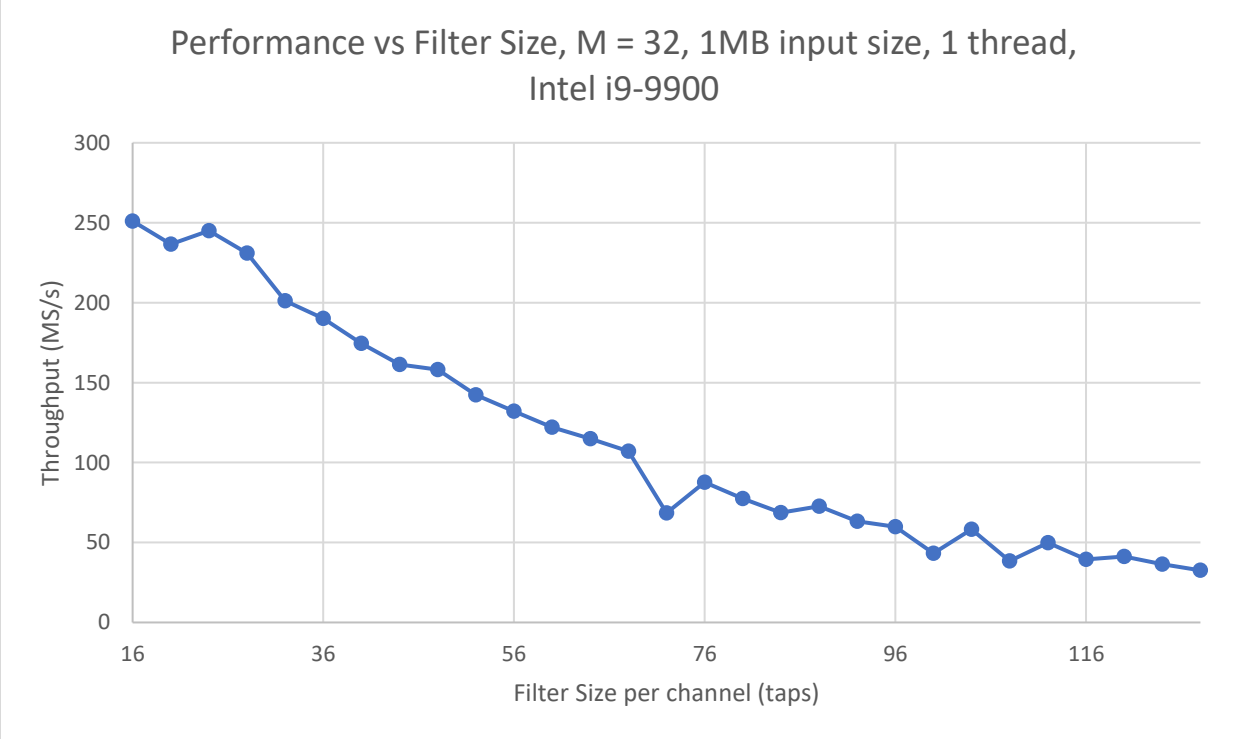


Figure 4: Performance is affected close to linear with filter size.

6 Usage

Examples are provided with the API. Each API function is documented in the shc_api.h header file.

6.1 Data Type

I/Q samples into and out of the channelizer are provided as interleaved complex floating-point samples. An array of I/Q values will have the form,

```
float myArray[N*2] = {Re1, Im1, Re2, Im2, …, ReN, ImN};
```

6.2 Creating the Channelizer

The user must first create a channelizer. To do this, users must select the following variables,

- M, Output channel count
- N, Number of output samples per channel to process per call to the API. The input size per API call will then be M*N complex samples. The input size in bytes is calculated as M*N*sizeof(complex float) or M*N*8.
- K, Filter size. Filter size must be a multiple of M. A filter size of 16*M is equivalent to running a 16-tap FIR filter at the decimated rate.
- Filter taps, These are the FIR filter taps. The FIR filter should be a standard low pass FIR filter with cutoff frequency below 0.5/M, where 0.5 represents the maximum low pass cutoff frequency for a complex low pass filter. The FIR filter must be real-valued. The API provides a function for generating these filter taps, or you can generate your own.
- Number of threads, Specifies the number of threads to be used for processing. If 1 thread is selected, processing occurs in the calling thread. When more than 1 thread is specified, a thread pool is created, and the calling thread waits until results are available.

The channelizer is created with the shcCreate() function. This function will return an integer handle that is used to interface this channelizer.

6.3 Output Format

Once the channelizer is created, you must choose an output format with the shcSetOutputFormat() function. The API provides two output formats, contiguous and non-contiguous. Output format specifies the expected memory layout for the output channels. Contiguous means the channels will be returned in one large array of length M*N complex samples. Non-contiguous means the user will provide an array of pointers to M, N length complex arrays, one for each channel.

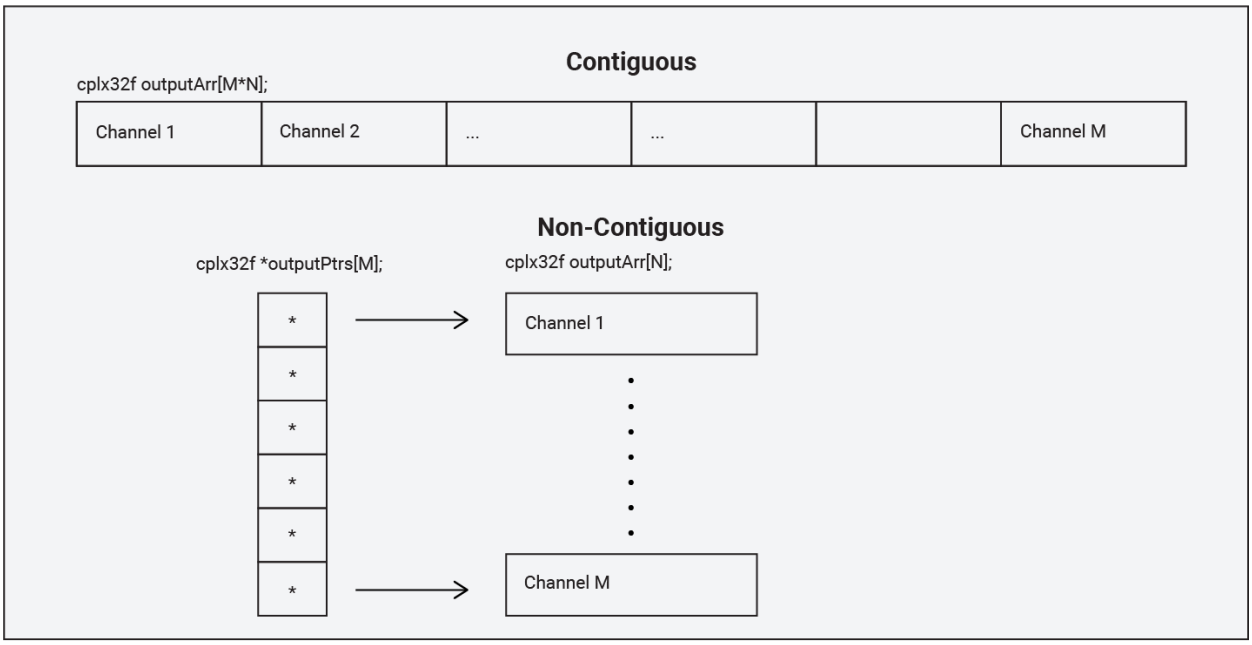


Figure 5: Illustrating the difference in memory layout for the 2 output formats.

6.4 Single-Threaded Interface

When the channelizer is operating as single threaded, users will call the shcProcess() function. This function will immediately process M*N input samples and return the results to the user.

6.5 Multi-Threaded Interface

When T threads are specified where T > 1, users will interface the API with the shcStart() and shcFinish() function. shcStart() queues M*N input samples. Users can queue up to T number of M*N input sample blocks, at which point shcFinish() must be called to retrieve the results from one item in the queue. The queue is FIFO.

For maximum throughput, users will queue T input blocks, and maintain T items in the queue by starting 1 queue item for every call to shcFinish().

See the multi-threaded examples for more information.

6.6 Notes

- The API is not thread safe. If accessing the API between multiple threads, use a mutex on all function calls.