Real-Time Digital Signal Processing with SciPy Signal:

Simultaneous Demodulation of Multiple FM Stations

Luigi Cruz
@luigifcruz

Software Defined Radios

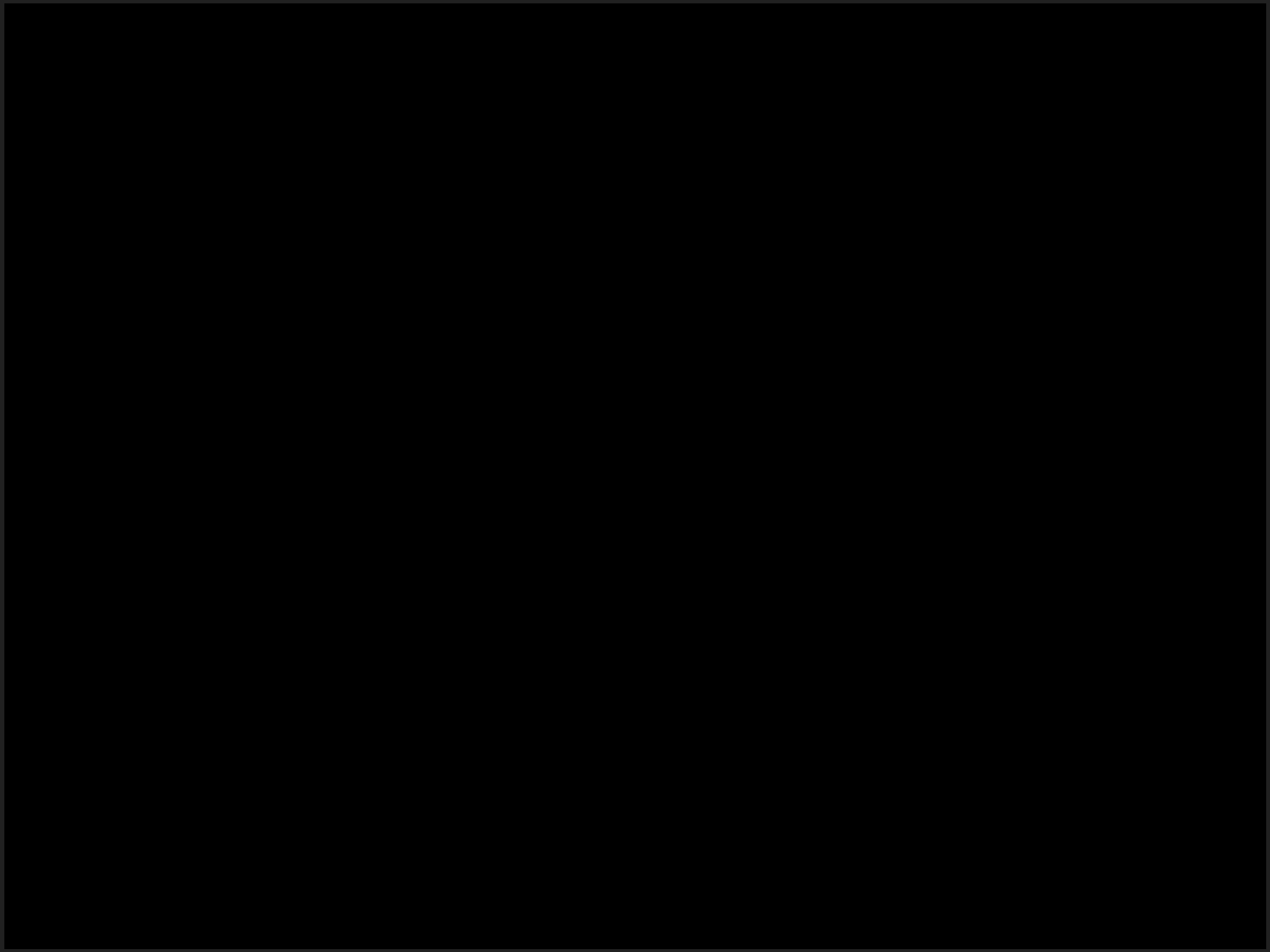
- Radio system implemented in software.
- Hardware responsible to tune to the right frequency and digitize the signal.
- The SDR outputs a stream of I/Q floating-point voltages to the computer.
- Can receive wideband signals like Digital TV, LTE, 5G and Wi-Fi signals.

FM Broadcast Demodulation

- Encodes information on a carrier wave varying in frequency.
- The encoded information can be recovered using a differentiator and envelope detector.
- Easily doable with SciPy Signal and Numpy.
- This generates a Mono output. More processing is necessary for Stereo.

```
_tmp = self._xp.angle(_tmp)
_tmp = self._xp.unwrap(_tmp)
_tmp = self._xp.diff(_tmp)
_tmp = self._xp.pad(_tmp, (1, 0))
_tmp = _tmp / self._xp.pi
_tmp = self._decimate.run(_tmp)
```

FM Broadcast Demodulation



Simultaneous Demodulation on the GPU

- An SDR receives the entire FM spectrum 88 MHz to 108 MHz (20 MHz).
- Channelizes the input into individual FM stations (200 kHz).
- Demodulate the FM Broadcast into a Stereo audio output (48 kHz).
- Audio can be saved on disk or analyzed on the GPU (e.g. ASR).

```python
@dataclass
class Config:
    enable_cuda: bool = False       # If True, enable CUDA demodulation.
    input_rate: float = 10e6        # The SDR RX bandwidth.
    device_name: str = "airspy"     # The SoapySDR device string.
    deemphasis: float = 75e-6       # 75e-6 for Americas and Korea, otherwise 50e-6.
    channels = [
        Channel(96.9e6, 240e3, 48e3, WBFM),
        Channel(94.5e6, 240e3, 48e3, MFM),
        Channel(97.5e6, 240e3, 48e3, FM),
    ]

```

Realtime Operations Optimizations for Python

What I learned from this project.

Better performance at no cost.

1 - Floating-Point Precision

- Wrong precision floating-points generates inefficiency.
- Impacts memory usage and processing performance.
- Be mindfull about the dtype of your array.
- *It's free real estate!*

PULL REQUEST #15366 (MERGED)

- Sometimes third-party functions cast the dtype into higher precision.
- Hibert functions were ~35% slower on the CPU with single-precision FP.

```
In [3]: data64 = np.random.rand(2**21).astype(np.float64)

In [4]: data32 = data64.astype(np.float32)

In [5]: %timeit sc.hilbert(data64)
126 ms ± 284 µs per loop (mean ± std. dev. of 7 runs, 10

In [6]: %timeit sc.hilbert(data32)
102 ms ± 481 µs per loop (mean ± std. dev. of 7 runs, 10
```

Original Code

```
In [3]: data64 = np.random.rand(2**21).astype(np.float64)

In [4]: data32 = data64.astype(np.float32)

In [5]: %timeit sc.hilbert(data64)
130 ms ± 259 µs per loop (mean ± std. dev. of 7 runs, 10 l

In [6]: %timeit sc.hilbert(data32)
75.6 ms ± 174 µs per loop (mean ± std. dev. of 7 runs, 10
```

Patched Code

1 - Floating-Point Precision

CUSIGNAL PULL REQUEST #447 (MERGED)

- Bigger difference on the GPU implementation.
- Hibert functions were ~87% slower on the GPU with single-precision FP.

Patch

In [10]:
```
%timeit -n 1000 gpu_sci.hilbert(gpu_data64)
```
```
2.46 ms ± 6.13 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [16]:
```
%timeit -n 1000 gpu_sci.hilbert(gpu_data32)
```
```
579 µs ± 16.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Upstream

In [3]:
```
%timeit -n 1000 gpu_sci.hilbert(gpu_data64)
```
```
2.45 ms ± 58 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [4]:
```
%timeit -n 1000 gpu_sci.hilbert(gpu_data32)
```
```
1.47 ms ± 64 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

2 - Threading with Audio

- Blocking calls are used by the SDR driver to transfer data to the application.
- This causes the execution to halt until new data is available.
- This dramatically reduces the time available for processing.
- It's important to NEVER block the audio thread.

SOLUTION

- Create one thread for processing and other of audio playback.
- Synchronize send data using ring-buffers.

3 - Ring Buffers

- A DSP program is a chain of discrete functions sharing vectors of data.
- Sometimes the consumer and producer are in different threads.
- The length of a vector can change between DSP functions.

SOLUTION

- Provides synchronization.
- Smoothly crosses these lenght boundaries.
- It allocates memory on initialization and reuse throughout the execution.

4 - Stop Repeating Work

- Underlying implementation might duplicate operations.
- Unnecessary processing depending on the method order.

EXAMPLE

- Resample function from SciPy Signal.
- Function expects input in the time-domain.
- If the data is in the frequency-domain, a conversion is required (iFFT).
- But the function will convert the input to frequency-domain internally (FFT).
- Useless operations!

scipy.signal.resample

`scipy.signal.resample(x, num, t=None, axis=0, window=None)` [source]

Resample *x* to *num* samples using Fourier method along the given axis.

The resampled signal starts at the same value as *x* but is sampled with a spacing of `len(x) / num * (spacing of x)`. Because a Fourier method is used, the signal is assumed to be periodic.

4 - Stop Repeating Work

PULL-REQUEST #11776 (MERGED)

scipy.signal.resample

`scipy.signal.resample(x, num, t=None, axis=0, window=None, domain='time')` #

[source]

Resample *x* to *num* samples using Fourier method along the given axis.

The resampled signal starts at the same value as *x* but is sampled with a spacing of `len(x) / num * (spacing of x)`. Because a Fourier method is used, the signal is assumed to be periodic.

```
+     if domain == 'time':
+         # Forward transform
+         if real_input:
+             X = sp_fft.rfft(x, axis=axis)
+         else:  # Full complex FFT
+             X = sp_fft.fft(x, axis=axis)
+     else:  # domain == 'freq'
+         X = x
```

Looking at the source code can help you achieve better performance!

5 - The GPU Likes Frequency-Domain Data

- Infinite Impulse Response (IIR) filters perform terribly on GPU.
- Every operation depends on the previous operation (can’t parallelize).

SOLUTION

- Finite Impulse Response (FIR) filters work on the frequency-domain.
- It’s parallelizable since operations don’t depend on each other.
- Implemented on cuSignal and SciPy Signal (firwin, lfilter, filtfilt).

scipy.signal.firwin

```
scipy.signal.firwin(numtaps, cutoff, width=None, window='hamming',
pass_zero=True, scale=True, nyq=None, fs=None)
```

[source]

FIR filter design using the window method.

This function computes the coefficients of a finite impulse response filter. The filter will have linear phase; it will be Type I if *numtaps* is odd and Type II if *numtaps* is even.

Type II filters always have zero response at the Nyquist frequency, so a ValueError exception is raised if firwin is called with *numtaps* even and having a passband whose right end is at the Nyquist frequency.

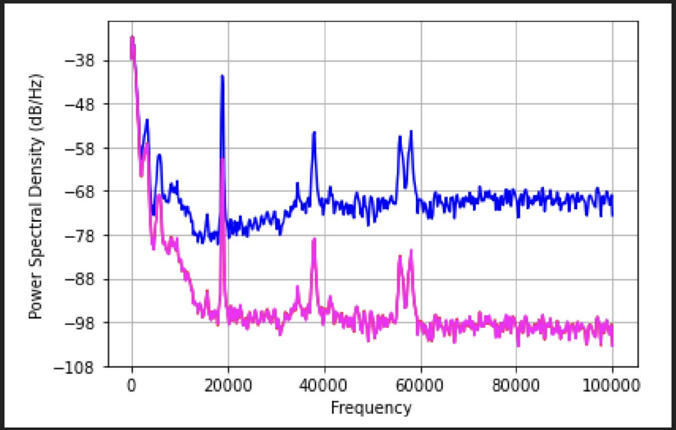
5 - The GPU Likes Frequency-Domain Data

IIR BASED FM-DEEMPHASIS

```
x = np.exp(-1/(200e3 * 75e-6))
b = [1-x]
a = [1, -x]
dsig1 = sc.lfilter(b, a, sig)
```

FIR BASED FM-DEEMPHASIS

```
butter = sc.dlti(b, a)
t, y = sc.dimpulse(butter, n=51)
coeffs = (np.squeeze(y), 1.0)
dsig2 = sc.lfilter(*coeffs, sig)
```



Fuchsia: FIR Filtered Signal

Red: IIR Filtered Signal

Blue: Original Signal

Thanks for listening!

EMAIL
contact@luigi.ltd

TWITTER & GITHUB
@luigifcruz

CONFERENCE SLACK
Luigi Cruz (@luigifcruz)

LINKEDIN
Luigi Cruz (@luigifc)

HAM RADIO CALLSIGN
PU2SPY

Radio Core Github

My Social Media