GPU Acceleration of DSP for Communication Receivers

Jake Gunther JAKE.GUNTHER@USU.EDU
Department of Electrical and Computer Engineering, 4120 Old Main Hill, Logan, UT 84322-4120 USA

Hyrum Gunther HYGUNTH@GMAIL.COM
Department of Electrical and Computer Engineering, 4120 Old Main Hill, Logan, UT 84322-4120 USA

Todd Moon TODD.MOON@USU.COM
Department of Electrical and Computer Engineering, 4120 Old Main Hill, Logan, UT 84322-4120 USA

Abstract

Graphics processing unit (GPU) implementations of signal processing algorithms can outperform CPU-based implementations. This paper describes the GPU implementation of several algorithms encountered in a wide range of high-data rate communication receivers including filters, multirate filters, numerically controlled oscillators, and multi-stage digital down converters. These structures are tested by processing the 20 MHz wide FM radio band (88-108 MHz). Two receiver structures are explored: a single channel receiver and a filter bank channelizer. Both run in real time on NVIDIA GeForce GTX 1080 graphics card.

1. Introduction

1.1. Background

On 28 October 2011, a Delta II rocket lifted off from Vandenberg Air Force Base carrying a weather satellite to a 820 km altitude, sun-synchronous orbit for NASA, NOAA, and DoD (NPP Launch). Along the way, the rocket deployed a pair of identical 1.5U (10 cm $\times$ 10 cm $\times$ 15 cm) cube satellites for the Dynamic Ionosphere CubeSat Experiment (DICE), a NASA/NSF sponsored mission (Fish et al., 2014). Each CubeSat was equipped with Languir probes to measure ionospheric plasma density, electric field probes to measure AC and DC electric fields, and a magnetometer to measure AC and DC magnetic fields. With slightly different orbital velocities, the CubeSats separated over time. Using two satellites, pairwise measurements could be used to resolve motion-induced ambiguities in observations of the ionosphere. Since variability in the ionosphere can dramatically interfere with radio frequency (RF) systems for communication and navigation, the ability of observe and predict conditions in the ionosphere is of great importance.

In addition to the science objectives, the DICE mission demonstrated a significant advancement in high-data rate communications capabilities over previous CubeSat missions. The science instruments aboard each DICE satellite are capable of generating approximately 1 Gbits of data each day. Assuming 7-10 minutes of overpass downlink times and 11 overpasses per day leads to the requirement of 1.5 Mbits/s downlink data rate. Prior CubeSat missions typically used amateur radio bands and equipment achieving 10 kbits/s. Thus the downlink data rates needed for DICE were 150 times greater than those used in most previous CubeSat missions. Each DICE satellite was equipped with a Cadet radio (originally built by L-3 Communications and now, as of 2017, provided by the Space Dynamics Laboratory). For the DICE mission, the Cadet used forward error correction (FEC) encoded OQPSK modulation and a downlink data rate of 3 Mbits/s in the 460-470 MHz band (UHF). The transmit power was between 1 W and 2 W.

The DICE ground station utilized a 18.3 m high-gain (approximately 35 dB) UHF dish at NASA's Wallops Flight Facility (WFF). The discrete-component RF equipment was minimal consisting of a low noise amplifier (LNA), a bandpass filter and cables leading to an Ettus USRP N210 (Ettus Research, a) containing a WBX daughter card (Ettus Research, b). The USRP fed baseband I/Q samples to a PC workstation. OQPSK demodulation and FEC decoding were performed using programs written in C.

The software defined nature of the receiver became a life saver for the DICE mission. When on-orbit operations began, the DICE telemetry data was demodulated and decoded at a low 10% success level. Investigations revealed the cause to be interference from the primary users of the 460-470 MHz band (police, emergency medical, pager systems, etc.). The space-to-Earth use is secondary in this band. When signal processing algorithms were developed and implemented to cancel intermittent narrowband inter-

Proceedings of the 7^{th} GNU Radio Conference, Copyright 2017 by the author(s).

ference from primary users, successful demodulation and decoding increased to a consistent 90-100% level (Gunther et al., 2015).

1.2. Problem: Slow processing

A problem that was never overcome during the DICE mission was the computational requirements of interference cancellation and demodulation. Despite best efforts at optimization, the C-programs for interference cancellation and demodulation could not run in real-time on the PC workstation. A work around was used throughout the DICE mission. The I/Q samples from the USRP were recorded to disk during the 7-10 minute overpasses. After the satellite disappeared over the horizon, the recording would be processed in a slower than real-time manner. This worked for DICE because overpasses were 90 minutes apart.

After the DICE mission, demand for the Cadet radio grew for CubeSat missions. The ground station code is available to the community through the Space Dynamics Laboratory. There remains a need to accelerate the implementation of the OQPSK demodulator to enable real-time processing during data downlinking.

1.3. Solution: Accelerating using GPUs

Recently the authors began to experiment with graphics processing units (GPU) as a means to accelerate digital signal processing algorithms such as those used in communication receivers. Our ultimate goal is to implement the interference cancellation and demodulation functions for the Cadet radio on GPU hardware and achieve faster than real-time execution of these functions. The demand for higher data rates is driving the need for acceleration even further.

A full OQPSK demodulator involves many different types of operations including filtering, multirate filters, phase locked loops (for carrier phase recovery and symbol timing recovery), automatic gain control, etc. As a first step toward a full OQPSK demodulator, this paper focuses on the implementation of filtering and multirate filtering operations on a GPU. To add realism to the problem we target the demodulation of broadcast FM signals. To add challenge to the real-time implementation, the full 20 MHz wide FM band (88-108 MHz) is processed. Two alternative implementations are explored. The first approach selects an FM station from the start and removes other stations through several stages of filtering and decimation (Oppenhime & Schafer, 2016). The second approach uses a uniform DFT filter bank (Vaidyanathan, 1993) to separate all FM stations simultaneously. These types of signal processing operations are useful in communication receivers. Future work will add additional processing elements to our growing library of GPU-based processing algorithms. In this paper, the USRP B205mini is used to acquire RF data.

1.4. Why GPUs?

Traditional CPUs are composed of a small number of cores (4 to 8) surrounded by large cache memories. This architecture can support a few software threads at a time. In contrast, a GPU is composed of hundreds of cores (100 to 1000 or more) and can support thousands of threads simultaneously. Thus GPUs offer the potential to accelerate software by factors of 100 or more compared to a CPU, provided that parallelization can be exploited. Acceleration via GPUs is more power and cost effective than comparable acceleration via CPUs.

Field programmable gate arrays (FPGAs) are often used to accelerate DSP algorithms. An advantage of GPUs over FPGAs is the ability to program GPUs using well known extensions of the C language such as CUDA (Sanders, 2011) and OpenCL (Scarpino, 2012).

1.5. Outline of the paper

The rest of the paper is organized as follows. Section 2 discusses FIR filtering as a basis for understanding our preferred GPU filter implementation. Section 3 extends the basic filter implementation to accommodate a reduction in the output sample rate yielding a structure for down sampling FIR filters. Extensions to multichannel signals is touched on briefly in Section 4. Together with a GPU-based numerically controlled oscillator, these structures are used to design, implement and test a real-time FM receiver as described in Section 5. This section also presents a filter bank approach that simultaneously channelizes all 100 FM stations. A summary and directions for future work are listed in Section 6.

2. Basic FIR Filtering

We begin with a discussion of time-domain FIR filtering. Later sections devoted to multirate filtering build on this discussion. Much of the literature on GPU-based filtering compare performance improvements over CPU-based implementations. In 2005, Smirnov and Chiueh compared time-domain implementations of FIR filters on GPU (Geforce 6600) and CPU (SSE-optimization on Pentium 4-HT 3.2 GHz) (Smirnov & Chiueh, 2005). At that time the SSE-optimized CPU was faster than the GPU except for very long filters. More recently Hirano and Nakayama (Hirano & Nakayama, 2010) and Rebacz, Oruklu and Saniie (Rebacz et al., 2010) showed that GPUs can outperform CPUs on FIR filtering problems.

2.1. Inner product view of convolution

The convolution formula

$$y_n = \sum_{k=0}^{N-1} h_k x_{n-k} \tag{1}$$

describing the input-output relationship of an FIR filter is well known. In (1) the sequence $h_0, h_1, \cdots, h_{N-1}$ is the filter impulse response, also know as the filter coefficients. As it is written, (1) has the appearance of an inner product (i.e. it is the sum of products). This viewpoint motivates the majority of FIR filter implementations such as the following C code example.

```
x[n] = input;
y = 0;
for(k=0; k<N; k++) {
  y += h[k] * x[(n+k)%N];
}
output = y;
n = (n - 1 + N) % N;
```

This example uses a circular data buffer with `n` being the circular index. If written out explicitly as the inner product of two vectors we have

$$y_n = \begin{bmatrix} \vdots \\ h_{N-2} \\ h_{N-1} \\ h_0 \\ h_1 \\ h_2 \\ \vdots \end{bmatrix}^T \begin{bmatrix} \vdots \\ x_{n-N+2} \\ x_{n-N+1} \\ x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \end{bmatrix},$$

where the filter coefficients are arranged so that h_0 is aligned with the current input sample x_n.

A straightforward implementation of an inner product in the multithreaded environment of a GPU requires N threads to multiply the elements of the vectors during the first time slot and requires $N/2$ threads and $\log_2 N$ time slots to do the adding in parallel. The efficiency is low due to the need for thread synchronization and because many threads sit idle in later time slots. It also requires accessing two arrays in memory.

2.2. Matrix-vector multiplication view of convolution

The matrix-vector multiplication view of convolution offers an alternative computational strategy that is more efficient in a multithreaded environment. Convolution is equivalent to multiplying a Toeplitz structured matrix by a vector. A matrix-vector product may be computed through computing inner products, and this view leads back to the standard inner product view of convolution. Alternatively, a matrix-vector product may be computed by a linear combination of the columns of the matrix. The scalars in the linear combination are the elements of the vector. Building on this idea, define the partial sum

$$y_n^m = \sum_{k=m}^{N-1} h_k x_{n-k}. \tag{2}$$

When $m = 0$ the partial sum becomes the complete convolution sum (1) and $y_n^0 = y_n$. Stacking these partial sums leads to the following relation at time n

$$\begin{bmatrix} \vdots \\ y_{n+N-2}^{N-2} \\ y_{n+N-1}^{N-1} \\ y_n^0 \\ y_{n-1}^1 \\ y_{n-2}^2 \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ y_{n+N-2}^{N-2} \\ y_{n+N-1}^{N-1} \\ y_n^0 \\ y_{n-1}^1 \\ y_{n-2}^2 \\ \vdots \end{bmatrix} + \begin{bmatrix} \vdots \\ h_{N-2} \\ h_{N-1} \\ h_0 \\ h_1 \\ h_2 \\ \vdots \end{bmatrix} x_n,$$

which is evidently a linear combination of vectors. Whereas the inner product approach maintains a circular buffer of input samples, the linear combination approach maintains a circular buffer of partial sums. After each sample is processed, the $m = 0$ accumulator is output and then reset to zero for the next iteration as in the following C code example.

```
for(k=0; k<N; k++) {
  y[k] += h[(k-m+N)%N] * input;
}
output = y[m];
y[m] = 0;
m = (m - 1 + N) % N;
```

The linear combination approach is advantageous in a multithreaded environment because the entire operation is completed in a single time slot. Each thread performs one multiply-accumulate operation and all threads work in parallel. This assumes that there is one thread for each coefficient in the filter. Each thread updates its own accumulator. It is memory efficient because only the array of filter coefficients is accessed. Furthermore, no threads sit idle. A CUDA kernel based on this idea follows.

```
__global__ void filt1(
    float* h, // coefficients
    float* y, // accumulators
    float* output, // output sample
    int* m, // time index
    float x) // input sample
{
    int j = threadIdx.x; // get thread index
    int k = (j - *m + N) % N;
```

```
    y[j] += h[k] * x; // multiply-accumulate
    if(k == 0) {
        *output = y[j]; // set output
        y[j] = 0; // reset accumulator
    }
    *m = (*m + 1) % N; // update circ. index
}
```

Here `N` is the length of the filter. If `N` threads are launched on the GPU and each one runs this kernel, each thread receives the input sample `x` and updates one element in the accumulator array `y`. The accumulator in one of the threads now holds a completed sum. That accumulator is output and its value reset for the next iteration. This kernel may be called from `main` as shown below.

```
int main(...) {
    ...
    while(cnt>0) {
        cnt=fread(&x,sizeof(float),1,fin);
        filt1<<<1,N>>>(d_h,d_y,d_out,d_m,x);
        cudaMemcpy(h_out,d_out,sizeof(float),
                   cudaMemcpyDeviceToHost);
        fwrite(h_out,sizeof(float),1,fout);
    }
    ...
}
```

This program reads input samples one-by-one from file. Each input sample is passed to one block of `N` threads running the `filt1` kernel on the GPU. The filter output is saved on the GPU in `d_out`. After the kernels finish, the output is copied back to the host CPU to the variable `h_out`, which is subsequently written to an output file.

Launching a kernel to process each input sample is not very efficient due to overhead and excessive memory copies. A more efficient approach is to copy a large chunk of input data to the GPU device, process the whole block, and then copy the output back to the CPU host. This reduces the number of kernel launches and memory copies. Furthermore, the chunk of input data can be broken up into overlapping slices and each slice can be processed on the GPU in separate thread blocks. This further exploits the parallelism inherent in the processing and also exploits the power of the GPU architecture.

Methods such as overlap-add and overlap-save for block convolution are well known (Oppenhime & Schafer, 2016) and will not be reviewed here. We adopt an overlap-save method in which the slices of the input chunk are overlapped by `N` samples, where `N` is the length of the filter impulse response.

```
__global__ void filt2(
    float* h, // coefficients
    float* y, // accumulators
    float* output, // array of outputs
    int* m, // time index
    float* x, // array of inputs
    int len) // length of slice
{
    int j = threadIdx.x;
    int start = len * blockIdx.x;
    int stop = len * (blockIdx.x + 1) + N;
    for(int i = start; i < stop; i++) {
        k = (j - m + N) % N;
        y += h[k] * x[i];
        if(i-N>=start) { // save valid samples
            if (k == 0) {
                output[i-N] = y;
                y = 0;
            }
        }
        *m = (*m + 1) % N;
    }
}
```

In this kernel, the overlap of input blocks is accomplished in the computation of `start` and `stop`. Notice that the `stop` index has an extra `N` samples added on. These samples overlap the first `N` samples of the next block. This kernel is called from `main` as follows.

```
int main(...) {
    ...
    while(cnt>0) {
        memcpy(h_x,h_x+xlen,N*sizeof(float));
        cnt=fread(h_x+N,sizeof(float),xlen,fin);
        cudaMemcpy(d_x,h_x,
                   (xlen+N)*sizeof(float),
                   cudaMemcpyHostToDevice);
        filt2<<<B,N>>>(d_h,d_y,d_out,d_m,d_x,
                       slice_len);
        cudaMemcpy(h_out,d_out,xlen*sizeof(float),
                   cudaMemcpyDeviceToHost);
        fwrite(h_out,sizeof(float),xlen,fout);
    }
    ...
}
```

Block convolution requires a lot of data movement between kernel calls as this example shows. The main point of this example is how parallelism is exploited. The large chunk of `xlen` input samples is broken down into `B` smaller slices of `slice_len` samples. Each of these smaller slices is processed in a separate block of threads. There are `B` blocks of threads with each block having `N` threads. The number of memory copies is drastically reduced over the previous example.

Block convolution is an approach to performing convolution on a long sequence by combining the results of convolutions on short subsequences. Block convolution is independent of the method used for convolution on the subsequences. Often the FFT is used to efficiently perform subsequence convolution. This has been explored in the setting of GPUs by Mauro (Mauro, 2012) and Belloch (Belloch et al., 2011). We did not pursue the FFT-based approach because the FFT loses efficiency when used for multirate filters, which is the subject of the next section.

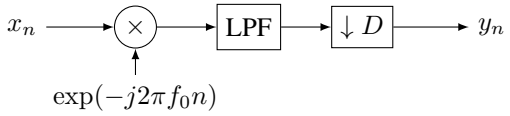

Figure 1. Block diagram of a digital down converter.

3. Multirate FIR Filtering

Digital down conversion (DDC) is prevalent in radio receivers. This operation translates the signal of interest to baseband and removes unwanted signals by low pass filtering. Down sampling is usually part of the DDC operation. The full operation is shown in Fig. 1. Upadhyay and Rajan (Upadhyay & Shakun Rajan, 2012) and Ma, Deng, Zhao (Ma et al., 2013) developed GPU-based DDC architectures and achieved significant processing acceleration compared to CPU implementations. Digital up conversion (DUC) is encountered in radio transmitters and essentially reverses all the operations in DDC.

The CUDA kernel for a down sampling FIR filter can be derived from the `filt2` example above with only a few minor modifications. Therefore, the code example is not given in this paper. Our code implementations are available in a GitHub repository (Gunther, 2017a).

4. Multichannel filtering

Direct conversion receivers (also known as zero-IF receivers) such as the AD9361 chip (Analog Devices, 2017) translates a frequency band of interest to baseband and provides digital I/Q samples. Typically the real and imaginary parts of a complex baseband signal are interleaved as is done with the USRP Universal Hardware Driver (UHD) (Ettus Research, 2017). The complex baseband signal is an example of a two channel signal. Antenna arrays consisting of multiple antennas and receivers output a multitude of channels that require subsequent processing through filters, DDCs, and correlators (Upadhyay & Shakun Rajan, 2012), for example. GPUs offer the programmable computational power needed to process multichannel signals. We have extended all the CUDA kernels described previously to operate on multichannel signals. However, publication page limitations do not permit a disclosure of those details in this paper.

5. FM Demodulator

Two approaches for FM demodulation were developed. In both cases, the entire 20 MHz wide FM band (88-108 MHz) was acquired using a USRP B205mini. The first approach uses the direct conversion architecture to acquire a single FM station. The second approach uses a filter bank to acquire all 100 FM stations simultaneously.

5.1. Direction conversion

Figure 2 shows a block diagram of the direct conversion receiver. Complex baseband samples from the USRP arrive on the left. The USRP acquires 20 MHz of bandwidth centered at 98 MHz and delivers 20 MSamples/s (MS/s) of interleaved real and imaginary samples to the host computer. Figure 3 shows the spectrum of the acquired signal as it is processed through the CUDA application.

The FM station of interest is tuned to baseband by selecting $f_0 = \frac{F_0 - 98}{20}$, where F_0 is an FM station in MHz. Each FM station is 200 kHz wide. The tuned signal may be down sampled by a factor of $20\text{MHz}/200\text{kHz} = 100 = 2 \cdot 2 \cdot 5 \cdot 5$. This is accomplished in four stages of filtering and down sampling as shown in Fig. 2. The filter pass and stop band frequencies are shown. In each case, the filters were designed to be equiripple filters with 0.1 dB of ripple in the pass band and 60 dB of stop band attenuation. Figure 3 shows signal spectra at several points in the system. The DDC isolates a single FM station and reduces the sample rate to 200 kS/s. These samples are passed to the quadrature FM demodulator shown in Fig. 2. The sample rate of the demodulated FM signal is reduced to 40 kS/s using a filter and decimate (by 5) stage.

This receiver is implemented using Unix pipes (FIFO) to transfer data between three separate C/C++ programs as shown in Fig. 4. The first program, labeled "UHD app." in Fig. 4, is a modified version of `rx_samples_to_file.c`, an example program included when UHD (Ettus Research, 2017) is installed. This program receives complex baseband samples from the USRP and writes them to a binary file. We modified this program to write samples to a pipe instead of a binary file. The second program, labeled "CUDA app." in Fig. 4 performs the following operations: (1) it reads complex baseband samples from the first pipe, (2) it uses CUDA to perform all the digital signal processing pictured in Fig. 2 (DDC and FM demodulation) on the GPU, and (3) it writes its real-valued output samples to the second pipe. The third program, labeled "JUCE app." in Fig. 4, reads samples from the second pipe and sends them to the sound card at 40 kS/s. This program uses JUCE (Storer, 2017), an open-source cross-platform C++ library for developing audio applications. Using pipes to move data between applications avoids the need to link UHD, CUDA and JUCE in a single executable and each program is automatically executed in separate process threads.

When multiple stages of filtering are performed on the GPU, it is advantageous to write intermediate signals to temporary buffers in the GPU global memory space. This avoids the need to copy intermediate signals between the host CPU and the GPU device. The input data is copied from the host to the device once. After that, kernels are

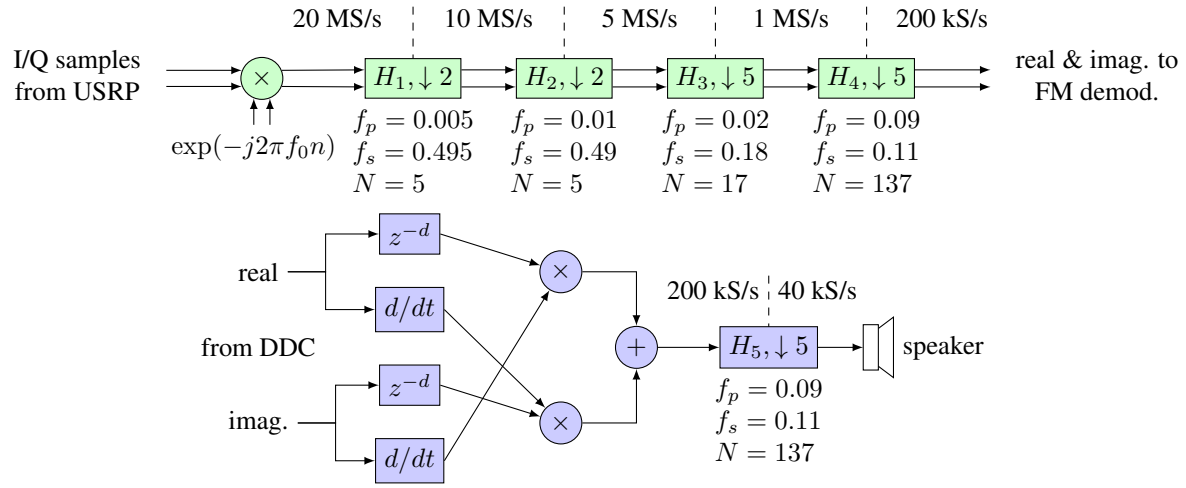

Figure 2. Block diagram of full FM receiver. I/Q samples enter the DDC (green boxes). Real and imaginary DDC output samples are fed to the quadrature FM demodulator (blue boxes). After FM demodulation, one stage of filtering and down sampling is used to reduce the sample rate to 40 kS/s.

Table 1. Execution Time for CPU and GPU Receiver Implementations

Number of Blocks	Chunk Size [$\times 10^6$]	CPU Time [sec]	GPU Time [sec]	Speedup
20	0.04	4.4147	0.9299	4.75
100	0.2	4.4096	0.2164	20.38
500	1	4.4255	0.0955	46.34
1000	2	4.4392	0.1054	42.12
2000	4	4.8956	0.1047	46.76
3000	6	4.4658	0.0987	45.25
4000	8	4.9637	0.1058	46.92
5000	10	4.9563	0.1091	45.43
10000	20	4.9796	0.1193	41.74

called sequentially to perform all the operations pictured in Fig. 2. Only the final audio output samples are copied back to the host CPU.

We compared the time required for GPU and CPU implementations of the entire processing chain shown in Fig. 2. Both were implemented in C. The GPU code base was compiled using `nvcc` which uses `gcc` to compile the host code. The CPU version of the code was compiled using `gcc`. Both versions were executed 10 times and mean execution time was recorded. Table 1 presents the results.

5.2. Filterbank channelizer

The FM receiver pictured in Fig. 2 demodulates a single FM station. If more than one station is desired, then the whole receiver structure must be duplicated for each desired station. When many stations are desired, there are more efficient structures to recover the stations of interest.

Figure 5 shows a uniform DFT filter bank. This structure uses a commutator to distribute incoming samples among a bank of 100 filters. This reduces the sample rate by a factor of 100 from 20 MS/s to 200 kS/s. After filtering on each channel, the 100-point FFT simultaneously and efficiently separates and isolates all 100 FM stations from one another. Then the stations of interest can be demodulated and the rest of the channels can be ignored. In the example of Fig. 5, channels 1, 3, and 4 are demodulated.

The filter bank demodulator relies upon the design of a prototype filter $H(z)$. The coefficients of this prototype filter are distributed among the 100 channel filters $H_0(z), H_1(z), \cdots, H_{99}(z)$. In our design, the prototype filter had 5100 coefficients so that the channel filters each had 51 coefficients. For more information on this type of filter bank see (Vaidyanathan, 1993). Our code implementation of the filter bank is available in a GitHub repository (Gunther, 2017b).

As the number of desired FM stations increases from 1 to 100, there is a point at which the filter bank demodulator becomes more efficient than duplicating the single channel FM receiver. With our implementation, we found that the break even point is when more than 5 channels are desired.

An example of the output of the filter bank channelizer is shown in Fig. 6. To display all 100 signals simultaneously, the complex magnitudes were computed for the signals over time and saved in a 2D array. This array is displayed as an image in Fig. 6. The color represents the signal magnitude is displayed in decibels. As shown in the picture,

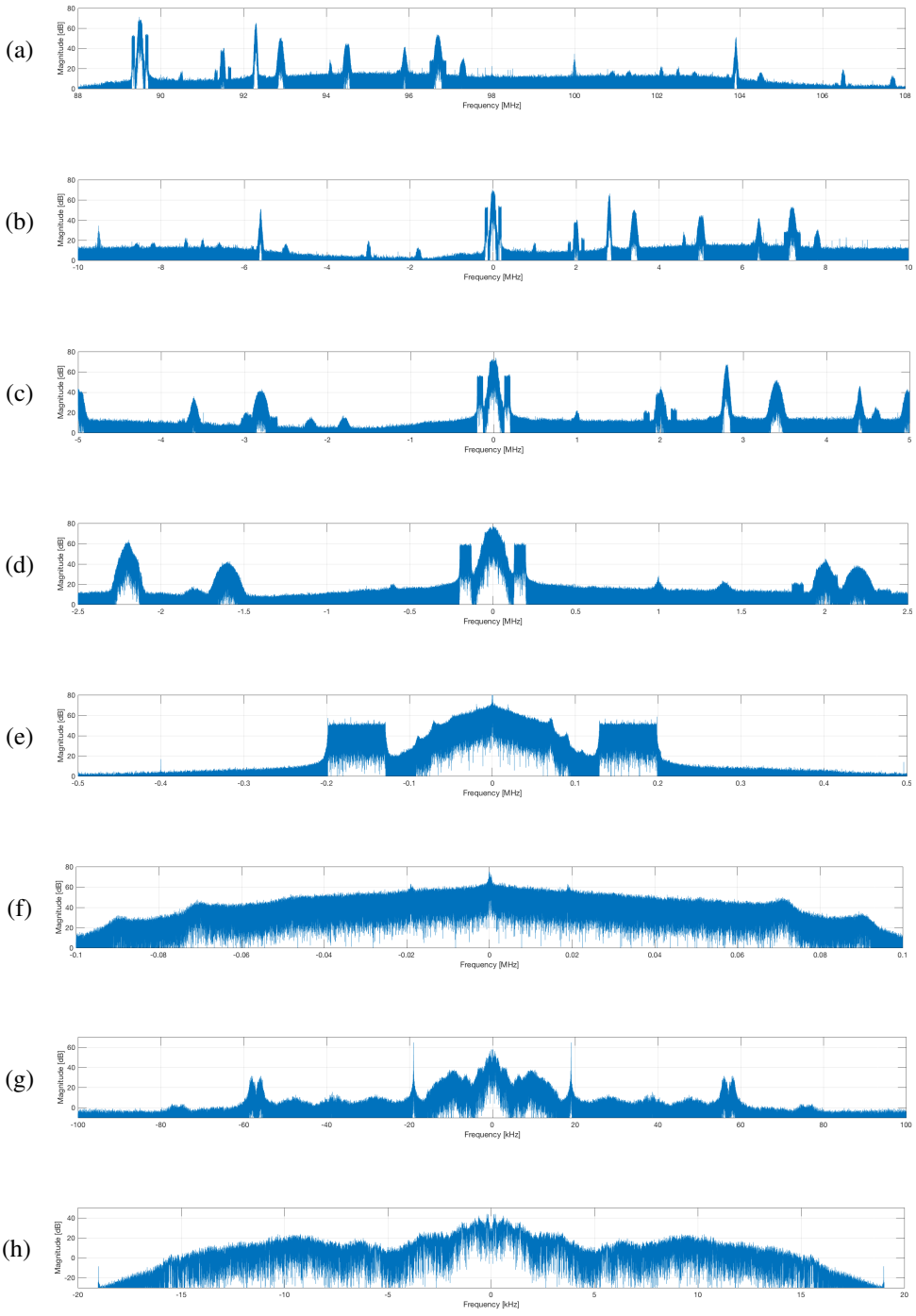

Figure 3. Signal spectra at various points in the processing of Fig. 2: (a) bandpass FM band, 88-108 MHz; (b) base-banded spectrum from USRP after tuning station of interest (89.5) to zero frequency; (c) after first stage of filtering and decimation by 2; (d) after second stage of filtering and decimation by 2; (e) after third stage of filtering filtering and decimation by 5; (f) after fourth stage of filtering and decimation by 5; (g) demodulated FM signal; (h) audio signal sampled at 40 kS/s.

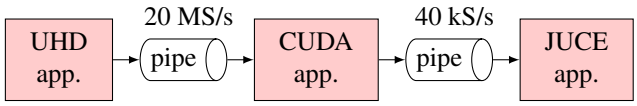

Figure 4. Software architecture for the FM receiver.

there are only about six strong FM stations received. Evidently there is no spectral leakage between channels. The strongest station appears to leak into neighboring channels, but this station broadcasts transmits HD Radio sidebands which explain the power present in adjacent channels.

6. Summary and Future Work

This paper presented an introduction to a growing library of signal processing algorithms for communication receivers. All algorithms are implemented in CUDA to exploit the computational power of GPUs. By exploiting parallelism and the power of GPUs, algorithms are capable of processing large swaths of bandwidth in real time. An example receiver in this paper consumed 20 MHz of bandwidth. To date the library of functions includes various types of filters including: basic FIR filters, pure delay filters, and differentiators. These filters are all available in multirate down sampling versions as well as multichannel versions. The filters can be cascaded together to create multistage processing chains. The library also includes complex mixing (for frequency translation) enabling digital down conversion. The library also includes routines for FM demodulation. By launching these kernels in multiple thread blocks, large blocks of input samples are processed in parallel using overlap save techniques. We plan to release our code library to the public as an online repository.

With these basic building blocks in place, we will turn attention to accelerating receivers for digital modulation formats. The feedback associated phase locked loops offers a challenge for parallelization.

References

Analog Devices. Rf agile transceiver. Technical report, Analog Devices, 2017. http://www.analog.com/media/en/technical-documentation/data-sheets/AD9361.pdf.

Belloch, Jose A., Gonzalez, Alberto, Martínez-Zaldívar, F. J., and Vidal, Antonio M. Real-time massive convolution for audio applications on gpu. *The Journal of Supercomputing*, 58(3):449–457, Dec 2011. ISSN 1573-0484. doi: 10.1007/s11227-011-0610-8. URL https://doi.org/10.1007/s11227-011-0610-8.

Ettus Research. USRP N210 web page, a. https://www.ettus.com/product/details/UN210-KIT.

Ettus Research. WBX web page, b. https://www.ettus.com/product/details/WBX.

Ettus Research. Usrp hardware driver and usrp manual. Technical report, Ettus Research, 2017. https://files.ettus.com/manual/.

Fish, C. S., Swenson, C. M., Crowley, G., Barjatya, A., Neilsen, T., Gunther, J., Azeem, I., Pilinski, M., Wilder, R., Allen, D., Anderson, M., Bingham, B., Bradford, K., Burr, S., Burt, R., Byers, B., Cook, J., Davis, K., Frazier, C., Grover, S., Hansen, G., Jensen, S., LeBaron, R., Martineau, J., Miller, J., Nelsen, J., Nelson, W., Patterson, P., Stromberg, E., Tran, J., Wassom, S., Weston, C., Whiteley, M., Young, Q., Petersen, J., Schaire, S., Davis, C. R., Bokaie, M., Fullmer, R., Baktur, R., Sojka, J., and Cousins, M. Design, development, implementation, and on-orbit performance of the dynamic ionosphere cubesat experiment mission. *Space Science Reviews*, 181(1):61–120, May 2014. URL http://dx.doi.org/10.1007/s11214-014-0034-x.

Gunther, Hyrum. Cuda-dsp. GitHub, Augsut 2017a. https://github.com/rumbonium/CUDA-DSP.

Gunther, Hyrum. Cuda-filterbank. GitHub, Augsut 2017b. https://github.com/rumbonium/CUDA-Filterbank.

Gunther, Jacob, Fish, Chad, Swenson, Charles, and Moon, Todd. Reliable space-to-earth communication as a secondary service in the 460470mhz band. *International Journal of Satellite Communications and Networking*, 33(2):93–106, 2015. ISSN 1542-0981. doi: 10.1002/sat.1072. URL http://dx.doi.org/10.1002/sat.1072.

Hirano, A. and Nakayama, K. Implementation of large-scale FIR adaptive filters on NVIDIA GeForce graphics processing unit. In *2010 International Symposium on Intelligent Signal Processing and Communication Systems*, pp. 1–4, Dec 2010. doi: 10.1109/ISPACS.2010.5704666.

Ma, Xiao, Deng, Lixia, and Zhao, Yuping. Implementation of a digital down converter using graphics processing unit. In *2013 15th IEEE International Conference on Communication Technology*, pp. 655–660, Nov 2013. doi: 10.1109/ICCT.2013.6820456.

Mauro, D. A. Audio convolution by the mean of GPU: CUDA and OpenCL implementations. In *Proceedings of the Acoutsics 2012 Nantes Conference*, 2012.

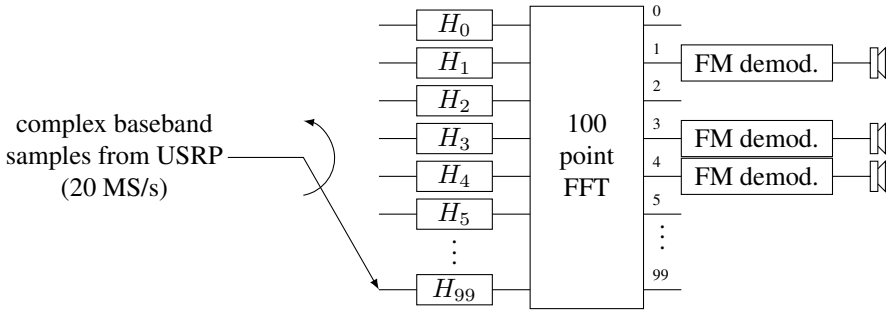

Figure 5. A uniform DFT filter bank channelizer is used to separate all 100 FM stations and down sample each to 200 kS/s. FM demodulation is applied to stations 1, 3, and 4.

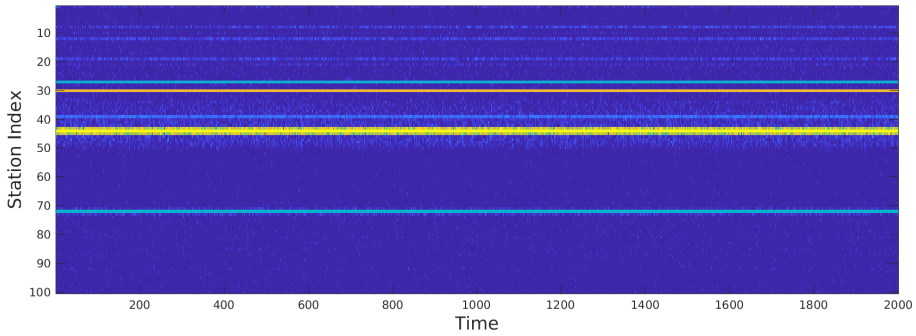

Figure 6. Magnitude of the filter bank channelizer output over time. Signals for all 100 stations are shown.

NPP Launch. Npp launch. https://www.youtube.com/watch?v=aCCW3D7fvn4&list=PL5DE2DDABD8EA7E97.

Oppenhime, A.V. and Schafer, R. W. *Discrete-Time Signal Processing*. Prentice Hall, 3rd edition edition, 2016.

Rebacz, J., Oruklu, E., and Saniie, J. Exploring scalability of FIR filter realizations on graphics processing units. In *2010 IEEE International Conference on Electro/Information Technology*, pp. 1–5, May 2010. doi: 10.1109/EIT.2010.5612114.

Sanders, Jason. *CUDA by example: An introduction to general-purpose GPU programming*. NVIDIA Corporation, 2011.

Scarpino, Matthew. *OpenCl in Action*. Manning Publications Co., 2012.

Smirnov, A. and Chiueh, T. An implementation of a FIR filter on a GPU. Technical report, Experimental Computer Systems Lab, Stony Brook University, 2005.

Storer, Julian. Juce. https://www.juce.com/, 2017.

Upadhyay, A. and Shakun Rajan, Y. Implementation of digital down converter in GPU. 2012.

Vaidyanathan, P.P. *Multirate Systems and Filter Banks*. Prentice Hall, 1993.