Experiences with using GNU Radio for *Real-time* Wireless Signal Classification

Christopher Becker CBECKER@CS.UTAH.EDU
Aniqua Baset ANIQUA@CS.UTAH.EDU
Sneha Kasera KASERA@CS.UTAH.EDU
University of Utah, 50 Central Campus Drive, Salt Lake City, UT 84112 USA

Kurt Derr KURT.DERR@INL.GOV
Samuel Ramirez SAMUEL.RAMIREZ@INL.GOV
Idaho National Laboratory, 2525 Fremont Ave, Idaho Falls, ID 83415 USA

Abstract

The ability to monitor the wireless spectrum in real-time is important in a variety of environments including high-security and control-system environments such as power plants and military facilities, as well as shared spectrum environments such as the 3.5 GHz band model that was announced by the Federal Communications Commission (FCC). In all of these cases, real-time detection and classification of signals while minimizing missed detections and misclassifications is paramount. Motivated by these important applications, we built a real-time system for spectrum monitoring and analysis which uses GNU Radio and Universal Software Radio Peripheral (USRP) X310s. In this paper, we focus on the GNU Radio-specific implementation challenges we face as well as the approaches we take to tackle these challenges. We also present our experiences with our implementation. We show that in some instances, particularly message passing, we can achieve a substantial improvement in processing performance by using alternative mechanisms, including Qt Signals and Slots (yielding a 78x performance improvement) and treating streams of data as strings, or by simply improving upon the existing code such as switching to using VOLK.

1. Introduction

The ability to monitor the wireless spectrum in real-time is important in a variety of environments including high-security and control-system environments such as power plants and military facilities, as well as shared spectrum environments such as the 3.5 GHz band model that was announced by the Federal Communications Commission (FCC) in rulings (FCC, April 2012) (FCC, July 2012)(FCC, April 21, 2015). In the case of high-security and control-system environments, only a limited number of signal types may be allowed for security reasons. These environments can be impacted in different ways: changes in the signals, either the addition of unauthorized signals or the removal of authorized signals, may indicate the presence of an adversary or a problem that must be addressed quickly to avoid possible consequences such as a breach or system failure; devices that do not act as intended or emit additional signals can cause interference or problems within the environment. In the case of the shared spectrum environment, a three-tier model has been proposed which provides different levels of service to different types of users: incumbent users (e.g., naval radars), priority access license (PAL) users (e.g., mobile service providers), and generalized authorized access (GAA) users which is controlled by a Spectrum Access System (SAS). Incumbent users are guaranteed to have the highest priority and interference-free access of the band. PAL users have prioritized access of the band when incumbents are absent. GAA users may use the band, but must not interfere with incumbent or PAL users and are not guaranteed an interference-free environment. Efficient mechanisms and protocols must be used for various tasks related to shared access enforcement: detecting the presence of incumbent transmitters, channel allocation among PAL users, and spectrum usage enforcement among PAL users and GAA users with the tiered access. The SAS must react quickly as the environment changes (e.g., an incumbent transmitter comes online) so as to provide the guarantees required by the different users. In all of these cases, real-time detection and classification of signals while minimizing missed detections and misclassifications is paramount.

Proceedings of the 3^{rd} *GNU Radio Conference*, Copyright 2018 by the author(s).

Motivated by these important applications, we built a real-time system for spectrum monitoring and analysis which we report in (Becker et al., Under Submission, 2018) which uses GNU Radio (GNU Radio) and Universal Software Radio Peripheral (USRP) (Ettus Research, 2017b) X310s. In that work, we focus on the system capabilities, the classification techniques used, and the classification results. In this paper, we focus on the GNU Radio-specific implementation challenges we face as well as the approaches we take to tackle these challenges. We present our system (Becker et al., Under Submission, 2018) briefly in Section 2. However, this system only serves as an example, but the challenges we identify can apply to other real-time, high sample rate tasks built on GNU Radio. The challenges we tackle in this paper include:

- GNU Radio message passing
- overflow and dropped packet detection
- non-optimized block implementations
- dynamic decimation
- the GNU Radio scheduler bottleneck
- no GPU acceleration

We evaluate the performance of our approaches. We show that in some instances, particularly message passing, we can achieve a substantial improvement in processing performance by using alternative mechanisms, including Qt Signals and Slots (yielding a 78x performance improvement) and treating streams of data as strings, or by simply improving upon the existing code such as switching to using VOLK. We also show that improving the performance is very challenging and that an approach that works in one instance will not always work in other instances so a variety of techniques, including combining blocks and alternative uses for built-in mechanisms, must be employed and tested.

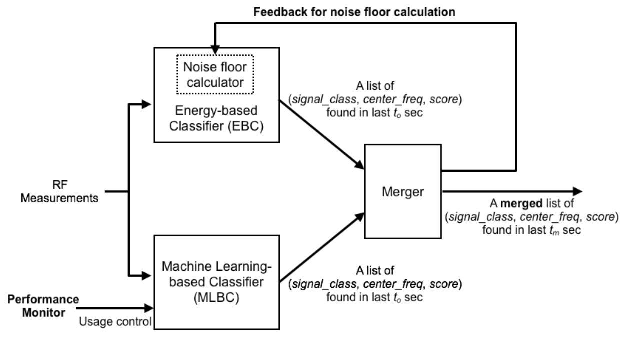
Feedback for noise floor calculation
Noise floor calculator
Energy-based Classifier (EBC)
A list of (signal_class, center_freq, score) found in last t_o sec
RF Measurements
Merger
A merged list of (signal_class, center_freq, score) found in last t_m sec
Machine Learning-based Classifier (MLBC)
A list of (signal_class, center_freq, score) found in last t_o sec
Performance Monitor
Usage control

Figure 1. System Architecture

2. System Overview

This section provides an overview of a subset of the system which we reported in (Becker et al., Under Submission, 2018). Again, while we provide this as a fully working example that incorporates the challenges we identify, we believe these challenges apply to other real-time, high sample rate systems using GNU Radio. Note that we are not claiming any contributions in regards to the system in Figures 1 and 2.

Figure 1 provides an overview of our real-time classification system. The system itself is implemented as two separate C++ applications using Qt4 (Qt) and GNU Radio. The first application runs our classification scheme while the second application monitors the performance of our classification scheme. The reasons for the second application are discussed in Section 3.2. Our classification scheme consists of a low-cost, but typically less accurate classification technique (Energy-based Classifier) which runs continuously and a compute-intensive, but more accurate classification technique that runs occasionally (Machine Learning-based Classifier). The results from the techniques are merged to form a final result. This final result is processed further on another machine as well as used to provide feedback to the low-cost classification technique to improve its accuracy.

Figure 2 provides an outline of the flowgraph used to implement our classification system in GNU Radio. A “UHD: USRP Source” configured with a sample rate of 25 MHz and using a tune request for the center frequency is used to provide samples to the flowgraph. The samples are first run through a Fast Fourier Transform (FFT) block before being sent to both classification paths.

The low-cost classification technique we use is an energy-based classifier. This classification path consists of a power spectral density (PSD) calculator, noise floor calculator, and multiple signal classifiers. Each signal classifier consists of multiple blocks: a dynamic cutoff calculator, peak detector, bandwidth and/or timing analyzer, and a pattern matcher. The PSD, noise floor calculator, and dynamic cutoff calculator are used to dynamically determine the noise floor cutoff to use for classification based on feedback provided by the merger. The peak detector and bandwidth and timing analyzers extract features that are then used by the pattern matcher to provide classification results.

The compute-intensive classification we use is a machine learning-based classifier. This classification path consists of a data reducer, a feature set calculator, and a signal classifier. The data reducer is used to reduce the number of samples sent to the rest of the path and therefore limit the amount of computation done by the path based on feedback from the external performance monitor. The feature set calculator computes the Spectral Correlation Function

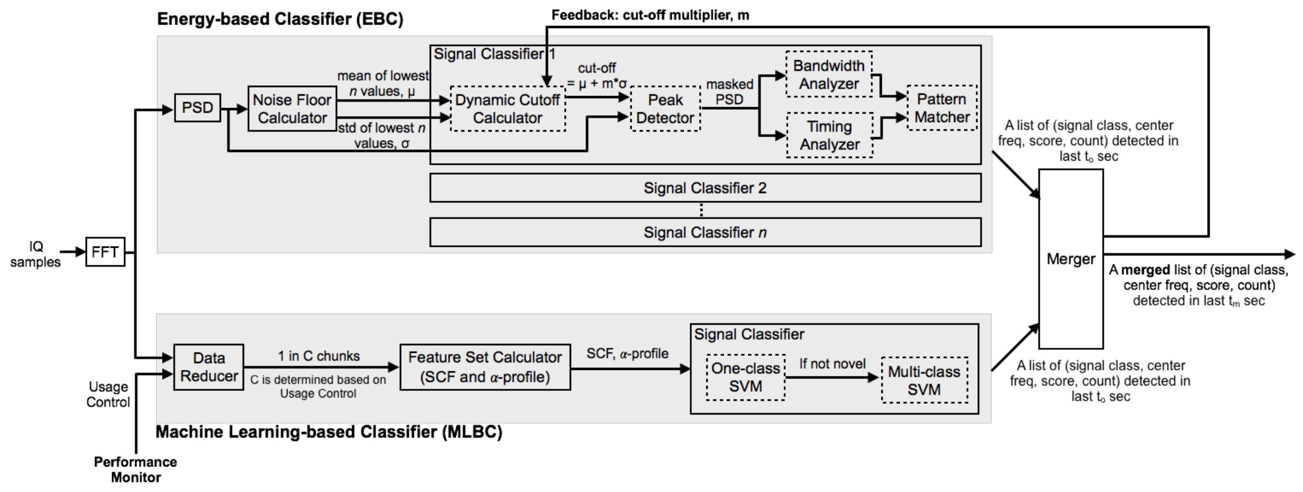
Energy-based Classifier (EBC)
Feedback: cut-off multiplier, m
Signal Classifier 1
IQ samples
FFT
PSD
Noise Floor Calculator
mean of lowest n values, μ
std of lowest n values, σ
Dynamic Cutoff Calculator
cut-off = μ + m*σ
Peak Detector
masked PSD
Bandwidth Analyzer
Timing Analyzer
Pattern Matcher
Signal Classifier 2
Signal Classifier n
A list of (signal class, center freq, score, count) detected in last t_o sec
Merger
A merged list of (signal class, center freq, score, count) detected in last t_m sec
Data Reducer
1 in C chunks
C is determined based on Usage Control
Feature Set Calculator (SCF and α-profile)
SCF, α-profile
Signal Classifier
One-class SVM
If not novel
Multi-class SVM
A list of (signal class, center freq, score, count) detected in last t_o sec
Usage Control
Machine Learning-based Classifier (MLBC)
Performance Monitor

Figure 2. Classification Flowgraph

(SCF) (Roberts et al., 1991) and α-profiles (Fehske et al., 2005) which are used as features for our machine learning models. The signal classifier module runs our pre-trained SVM (Cristianini & Shawe-Taylor, 2000) models on the feature set to determine the signal class and then forwards the results to the merger.

The merger takes classifications results from the two classification paths to determine final classification results for the classifier. These results are used to determine feedback to provide to the energy-based classifier for adjusting the noise floor cutoff calculation. The results are also sent to another system for further processing. Currently, the results are sent every two seconds, giving us under two seconds to perform the classification and send the results.

Due to our extensive use cases, our goal is to provide all this functionality on a system that does not require external network access and is somewhat mobile. Therefore, we target being able to run this part of our system on a high-powered laptop. This is also a constraint in terms of resources that we can use, forcing us to focus on optimizing performance whenever possible.

3. Challenges and Approaches

In the following subsections, we discuss the challenges we face while implementing our classification system using GNU Radio and the approaches we use to tackle these challenges.

3.1. Message Passing

One of the challenges we face is limitations with the message passing capabilities included within GNU Radio. Within GNU Radio, messages can be passed by either adding tags to streams of data or by sending messages between blocks. In the energy-based classifier path, data about signals is sent between the peak detector and bandwidth and timing analyzer blocks for feature extraction. Since we no longer need the raw samples after the peak detector, adding tags to a stream of data we no longer needed is inefficient. However, we quickly find out by "asynchronous message buffer overflowing, dropping message" warnings that we are sending more data than the GNU Radio messages framework can handle.

We use a variety of approaches to bypass the limitations of the message passing capabilities included within GNU Radio. In some cases, we use standard streams as fixed-sized formatted data strings to pass information between blocks. As one example, between two of our blocks, byte strings are used to represent true and false values. As another example, between other blocks, we use fixed-sized integer strings where non-used values of the stream are set to -1 to denote the end of useful information. While this approach is probably less useful in general blocks as the stream format needs to be clearly defined and some processing overhead is incurred in processing the streams, using streams in alternative ways can be an excellent approach between custom purpose-driven blocks that require data to be sent rapidly between each other.

In other cases, we use the Signals and Slots mechanisms in Qt to pass information between blocks that did not involve the overhead of encapsulating and unecapsulating the data. In order to keep our options open as far as using the blocks for both GNU Radio Companion (GNU Radio Companion) and within our own applications, blocks that use the Signals and Slots functionality are implemented as Qt Objects with thin GNU Radio block wrappers around them. While used in GNU Radio Companion, the GNU Radio block wrappers and GNU Radio's message passing mechanisms are used. However, both the GNU Radio blocks and the underlying Qt Objects are accessible in both C++ and Python code, allowing flexibility in usage particularly when used outside of GNU Radio Companion. For example, between the merger block and the dynamic cutoff calculator block, this approach is taken to pass changes given as feedback to the energy-based classification path.

As an approach similar to the Signals and Slots mechanisms in Qt, in some cases callback functions are used to move data as well. In particular, we use this approach to connect the thin GNU Radio wrappers to the underlying Qt Object block implementations. We provide an evaluation of the performance difference between passing data using GNU Radio's message passing capabilities and using Qt's Signals and Slots mechanism in Section 5.1.

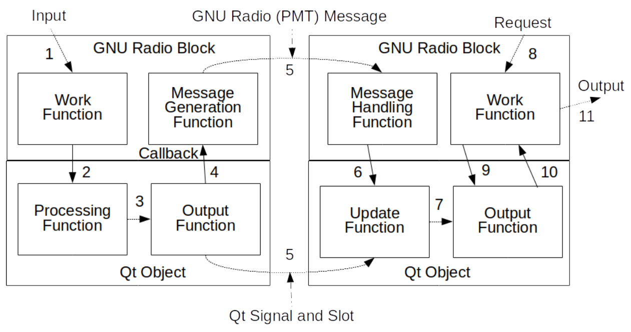

Figure 3. GNU Radio/Qt Object Blocks

Figure 3 provides an example overview of two Qt Object classes (bottom portions) with thin GNU Radio block class wrappers (top portions). The classes on the left take GNU Radio data streams and create asynchronous messages based on the input. The blocks on the right receive asynchronous messages and produce GNU Radio data streams based on the messages. The following steps are performed:

1. GNU Radio passes the input stream(s) to the GNU Radio block's work function
2. The work function passes the data to the internal QT Object's processing function
3. The processing function prepares data to be forwarded to other blocks via either GNU Radio Messages, Qt Signals and Slots, or a combination of the two
4. The output function calls the GNU Radio block's message generation function when output is ready and data is converted to GNU Radio message
5. Data is forwarded via the appropriate mechanisms to the next block
6. Data is converted from GNU Radio message to data (if necessary)
7. Internal data is updated and prepared for output
8. GNU Radio requests the block produce output
9. The work function passes the output buffer to the internal Qt Object's output function
10. The Qt Object's output function fills the buffer and returns control to the work function
11. The work function returns the output to GNU Radio

3.2. Overflow and Dropped Packet Detection

One of our longest-lasting challenges has been detecting overflows and dropped packets at run-time so that we can adjust the performance of our system dynamically. While the UHD source block will tag a stream with a new timestamp when an overflow or dropped packet occurs, no other information about the event is provided. Likewise, the UHD driver sends information about overflows and dropped packets to the stderror output and does not provide the details via its API in a way that is accessible to GNU Radio flowgraphs. Without this information, it is very difficult to maximize the performance of the flowgraph with the computational overhead versus classification accuracy trade off we face, particularly when the system resources may vary due to other processes starting or stopping or changes in processing requirements by other processes.

Performance counters, such as those described in (Rondeau et al., 2013) are useful tools during development, but we find them not as useful or as informative as we would like during run-time. To handle detecting overflows and dropped packets, we develop a separate program which takes piped stderror output from our classification program as well as information from the operating system to determine whether or not any overflow or dropped packets occur or if they are likely to occur. The program parses the stderror output from the classification program and looks for the number of "O's" and "D's" coming from the UHD driver as a metric to determine overflow and dropped packet rates. It then uses that information along with other information from the underlying operating system (e.g., network buffer usage, CPU usage, etc.) to send signals to the classification system to reconfigure itself and change the portion of samples being sent through the data reducer block to the machine learning-based classification path. By changing the number of samples being sent to our more compute-intensive part of our flowgraph, we can adjust to the changes in the operating environment as the work in the

Listing 1. ‘add_cc’ block work function for complex data type after template expansion

```
gr_complex *optr =
    (gr_complex *) output_items[0];
int ninputs = input_items.size ();
for (size_t i = 0;
     i < noutput_items*d_vlen;
     i++){
    gr_complex acc =
        ((gr_complex *)
        input_items[0])[i];
    for (int j = 1; j < ninputs; j++)
      acc +=
        ((gr_complex *)
        input_items[j])[i];
    *optr++ = (gr_complex) acc;
}
return noutput_items;
```

Listing 2. ‘add_cc’ block work function for complex data type using VOLK

```
gr_complex *out =
    (gr_complex *) output_items[0];
int noi = d_vlen*noutput_items;
memcpy(out,
       input_items[0],
       noi*sizeof(gr_complex));
for(size_t i = 1;
    i < input_items.size();
    ++i){
  volk_32fc_x2_add_32fc(out, out,
    (const gr_complex*) input_items[i],
    noi);
}
return noutput_items;
```

blocks will be completed quicker and the blocks will not be called as often by the scheduler.

3.3. Non-optimized Block Implementations

While attempting to optimize the performance of our flowgraph, we find some of the standard blocks within GNU Radio to not always be the most optimized implementations, at least for our specific use cases. For example, the standard ‘add_cc’ block does not take advantage of VOLK to add the samples together rather than iterating over each individual sample. The difference between the standard and VOLK implementations is shown in Listings 1 and 2 respectively. Another example is the log power FFT block. This block is a hierarchical block which uses a combination of ‘stream_to_vector_decimator’, ‘fft_vcc’, ‘complex_to_mag_squared’, ‘single_pole_iir_filter_ff’, and ‘nlog10_ff’ blocks to calculate the result. This block takes samples, computes the FFT internally and then calculates the PSD from those values using non-optimized code (i.e., does not take advantage of VOLK to speed up processing). However, in our case, we had to calculate the FFT for both classification paths, but only used the PSD for one of the paths, which meant we would have added additional computational overhead by using the standard blocks.

We reimplement several standard blocks to have them use VOLK or better serve our needs, reduce overhead, and provide better performance. These blocks are: ‘add_cc’, ‘fft’, log power FFT, and ‘analog_const_source’. We switch the add_cc block to using VOLK as shown above. We reduce the fft block overhead slightly and change it to better serve our interests. For example, we change the input to allow streamed input similar to the log power fft block. Changing the input also allows us to reduce our flowgraph by a block which can be important as described in Section 3.5. We break down the log power FFT as described above and change to using VOLK when possible. We change the analog_const_source block so we can change the value it is sending dynamically via messages. We evaluate the performance differences in Section 5.2.

3.4. Dynamic Decimation

One of our more recent challenges encountered while working on interaction between our performance monitor and flowgraph is dealing with dynamic decimation values. Our data reducer block, which allows us to adjust the amount of processing we are doing, has a constant decimation rate between commands from the performance monitor to change the amount of samples sent to the machine-learning based classifier. However, even while pausing and reconfiguring the flowgraph, we ran into issues with the buffer sharing methods used by GNU Radio. When the flowgraph would be restarted, we would get errors from GNU Radio that the input buffers to the block did not contain enough samples to be processed.

We find that by deleting elements in the flowgraph that change decimation, as well as the immediately preceding blocks, and re-initializing them while reconfiguring the flowgraph we are able to bypass the issues with the buffers. By deleting multiple blocks and re-initializing them, buffers are also reinitialized to the correct sizes. However, this also results in a loss of data due to the samples in those buffers being lost. We find that being able to have the flowgraph adapt to outside influences (e.g., system processing changes) quickly can be very useful.

3.5. Scheduler

Our most recent challenge encountered is when we run our flowgraph with a large number of blocks/threads, we actually see the thread used by the GNU Radio scheduler seem to become a bottleneck for the flowgraph. We monitor the processing load of individual threads within a process using the *top* command in Linux. Our application consists of roughly 40 threads. While monitoring the thread processing load, we see the thread containing the scheduler staying at a consistent 100% CPU usage, while the threads for the blocks themselves utilizing very little CPU usage and the UHD driver reporting overflows and dropped packets.

Having the GNU Radio scheduler be a bottleneck is still an open challenge for us. Currently, we have started trying to reduce the number of threads by reducing the number of blocks. If we reduce our thread count by as little as one thread, the CPU usage seems to spread out across the processors and stay in the 70-80% utilization range, underutilizing the processor. In many cases we combine the functionality of multiple standard blocks into a smaller set of newly-implemented blocks. In other cases, we change blocks to accept multiple inputs rather than limiting it to one input. In both cases, this has the effect of having fewer blocks that each do more work. When we run the flowgraph using these new blocks, the GNU Radio scheduler thread does not seem to be as much of a limiting factor to throughput. An evaluation involving combining blocks and adding inputs to blocks is provided in Section 5.3.

3.6. GPU Acceleration

We also explore using GPU acceleration, as explored by works such as (Gunther et al., 2017), (Hitefield & Clancy, 2016), and (Piscopo, 2017), for increasing the performance of our classification system, particularly one part of the machine-learning based classifier. We implement the feature set calculator (SCF and α-profile) block (as shown in Figure 2) as a CUDA kernel wrapped in a GNU Radio block. After several attempts at optimizing the implementation however, we come to a similar conclusion as (Hitefield & Clancy, 2016) and (Piscopo, 2017): the current implementation of GNU Radio does not lend itself well to GPU acceleration due to the limited data buffer sizes and the memory copying overhead for the blocks we are using.

In the end, we abandon the GPU acceleration approach and take the approach of using VOLK whenever possible as well as optimizing our own code as needed by various methods. Due to the amount of memory we have access to on our computers, we tend to focus on processing optimization rather than memory optimization. Focusing on processing optimization over memory optimization allows us to use constructs such as heaps and lookup tables to quickly find and manipulate internal data within our blocks while leading to slightly larger memory usage.

Table 1. Laptop Configurations

	Development and Testing	Testing and Evaluation
Operating System	Ubuntu 14.04.1 (LTS)	Ubuntu 18.04 (LTS)
UHD Version	3.11.0	3.11.1
GNU Radio Version	3.7.11	3.7.12
LibVOLK Version	1.3.0	1.4.0

Table 2. X310 Configurations

	Development and Testing	Testing and Evaluation
Revision	6	8
Firmware Version	5.1	6
FPGA Version	33	35

4. Setup

For all of our work, we use Dell M4800 laptops with increased RAM (2.8 GHz, 4-core 64-bit Intel i7-4810MQ processor and 32 GB of RAM) and USRP X310s using a 1 Gbps Ethernet Link, providing us with a sample rate of 25 MHz. Table 1 provides an overview of the laptop configurations we use. In both cases, the build GNU Radio script (GNU Radio Build Script) (slightly modified to work with Ubuntu 18.04) is used to install GNU Radio and its dependencies, including USRP Hardware Driver (UHD) (Ettus Research, 2017a) and LibVOLK (VOLK). Table 2 provides an overview of the X310 configurations we use.

5. Evaluation

In this section we evaluate some of the performance improvements made by the approaches used to meet the challenges we face in areas of Message Passing, Non-optimized Implementations, and the Scheduler as described in Sections 3.1, 3.3, and 3.5 respectively. For the tests, we use 4096 samples as the target number of samples due to the default behavior of buffers and the scheduler in GNU Radio as discussed in (Piscopo, 2017).

5.1. Message Passing

We test the performance of sending messages through the message passing mechanisms built into GNU Radio compared to using the Signals and Slots mechanism included in Qt. For this test, we implement two blocks that can either use GNU Radio's message passing mechanisms or the Qt Signals and Slots mechanisms. The first block receives a stream of integers and forwards them (as long integers) to

Table 3. Median Time to process 4096 outputs for various implementations

Implementation	Median Time
Standard Complex Add	0.00143647193909
VOLK Complex Add	0.00141096115112
Log Power FFT	0.0159084796906
PSD	0.0158760547638

the other block via one of the two mechanisms. The median time for 100 trials to process 4096 samples using GNU Radio's message passing mechanisms was 0.263926029205 seconds while the median time for the same setup using Qt's Signals and Slots mechanism was 0.00337398052216 seconds (more than a 78x performance improvement).

5.2. Non-optimized Implementations

We test the performance result of reimplementing some of the standard blocks to be more optimized or better suit our needs. Table 3 provides an overview of the median time it took to process 4096 samples for the different implementations. To get the median time, we performed the tests 100 times. For these tests, we compare the standard Complex Add block with our new VOLK Complex Add block as well as the standard Log Power FFT block with our PSD block which uses VOLK. As can be seen, in both cases our new implementations provide better performance. This performance gain can be substantial in the long run, particularly at high sample rates as well as when combined with other performance improvements made in other areas.

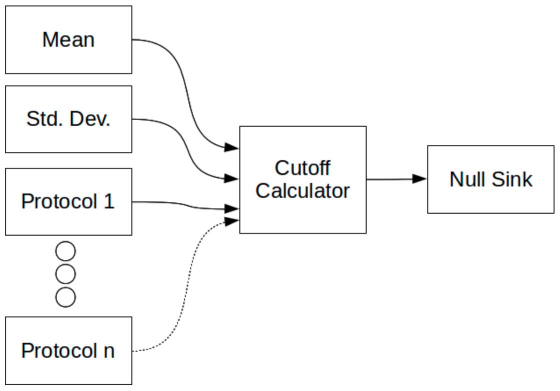

Figure 4. Flowgraph for Single Cutoff Calculator Timing Test

5.3. Combining Blocks or Adding Inputs

We test the performance result of reducing the number of blocks through combining blocks and/or adding additional inputs to the same block. For these tests, n represents the number of wireless protocols our classification system

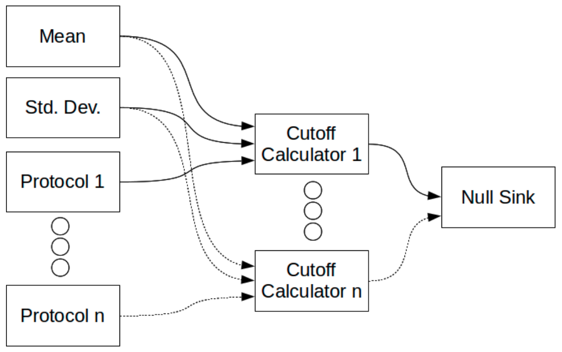

Figure 5. Flowgraph for Multiple Cutoff Calculators Timing Test

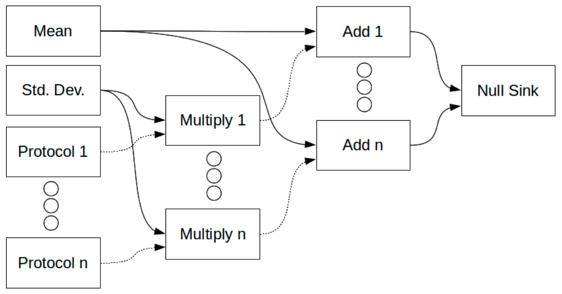

Figure 6. Flowgraph for Add and Multiply Timing Test

would be classifying against.

Figures 4, 5 , and 6 give an overview of the flowgraphs used for our tests. In these tests, we compare our custom cutoff calculator block and standard add and multiply blocks. In each setup, one 'vector_source' block is used to represent each protocol as well as two additional vector_source blocks to represent the mean and standard deviation used for the noise floor cutoff. Using the standard add and multiply blocks, this means we have $3*n+3$ blocks in play (n add blocks, n multiply blocks, $n+2$ vector_source blocks, and 1 null sink), resulting in $3*n+3$ threads. However, using our custom cutoff calculator block, we compute the same results using either $2*n+3$ or $n+4$ blocks ($n+2$ vector_source blocks, 1 or n cutoff calculators, and 1 null sink), resulting in $2*n+3$ or $n+4$ threads. Figure 7 shows the median time it took to get 4096 samples through our new custom cutoff calculator block versus completing the same operations using standard add and multiply blocks. Each test was run 50 times to determine the median. As can be seen, as more protocols are used, the larger the performance increase by reducing the number of blocks. The number of protocols our system will classify against will vary depending on the application, anywhere from less than 10 in the case of most licensed bands, to potentially 10s to 100s in the case of an unlicensed (e.g., ISM) band.

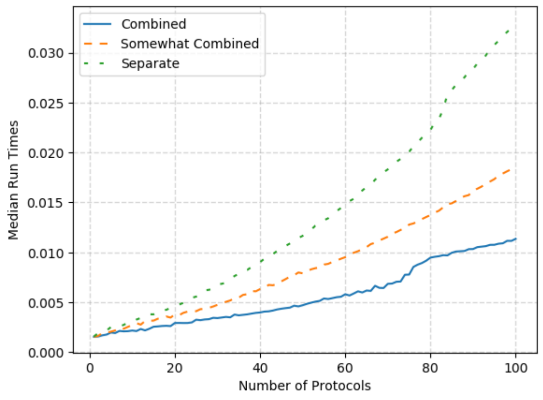

Figure 7. Median time to run 4096 samples through a single custom block (combined), multiple custom blocks (somewhat combined), and separate add and multiply blocks (separate)

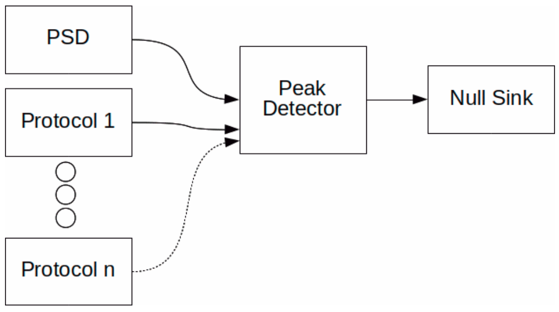

Figure 8. Flowgraph for Combined Block Input Test

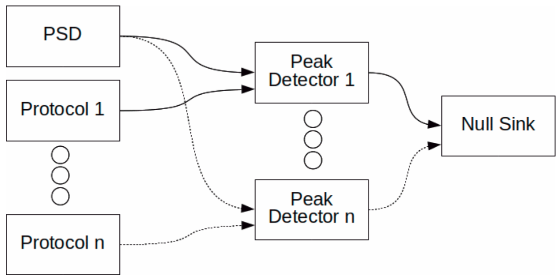

Figure 9. Flowgraph for Separate Block Input Test

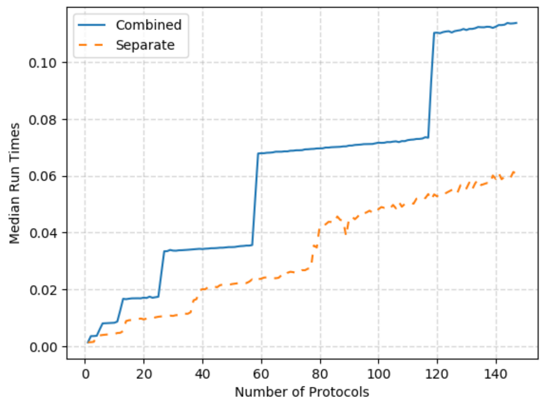

Figure 10. Median time to run 4096 samples through one block with multiple inputs versus multiple blocks

However, it is important to note that this performance increase is not always the case. Another test we run involves reducing the number of blocks by increasing the number of inputs that block will handle. Figures 8 and 9 give an overview of the flowgraphs used for a single peak detector with more inputs and multiple peak detectors. In each setup, one 'vector_source' block is used to represent the PSD data as well as n vector_source blocks to represent the values used for each protocol, resulting in $n + 3$ or $2 * n + 2$ threads respectively. Figure 10 shows the median time it took to get 4096 samples through our peak detector block. As can be seen, in this case, the performance actually decreases when adding more inputs to the block without converting it to a multi-threaded version. We are still investigating the step-like appearance, particularly in the case of the single peak detector. However, at this time it appears to correlate with increases with data cache misses in the vector_source and peak detector block seen using cachegrind (Cachegrind). Similar to the discussion in (Piscopo, 2017), different approaches are sometimes needed depending on the type of operations being performed within a block.

6. Conclusion

In this paper, we presented our experience with using GNU Radio and the USRP X310 to build a real-time spectrum monitoring and analyzing system. We identify important challenges we experienced using GNU Radio to implement our real-time system for spectrum monitoring and analysis as well as the approaches we took to tackle the challenges. We also evaluated the performance of some of our approaches. We showed that in some instances, particularly message passing, we can achieve a substantial improvement in processing performance by using alternative mechanisms, including Qt Signals and Slots (yielding a 78x performance improvement) and treating streams of data as strings, or by simply improving upon the existing code such as switching to using VOLK. We also showed that improving the performance is very challenging and that an approach that works in one instance will not always work in other instances so a variety of techniques, including combining blocks and alternative uses for built-in mechanisms, must be employed and tested.

References

Becker, Christopher, Baset, Aniqua, Kasera, Sneha, Derr, Kurt, and Ramirez, Samuel. Robust flexible system for real-time spectrum monitoring and analysis, Under Submission, 2018.

Cachegrind. Valgrind: Tool Suite. http://valgrind.org/info/tools.html#cachegrind, 2018.

Cristianini, Nello and Shawe-Taylor, John. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

Ettus Research. UHD. https://kb.ettus.com/UHD, 2017a.

Ettus Research. The Universal Software Radio Peripheral. https://www.ettus.com/product, 2017b.

FCC. In the Matter of Unlicensed Operation in the TV Broadcast Bands: Third memorandum opinion and order. FCC Document 12-36, April 2012.

FCC. Amendment of the Commissions Rules with Regard to Commercial Opeations in the 3550-3650 MHz Band, Report and Order and Second Further Notice of Proposed Rulemaking. FCC Document, April 21, 2015.

FCC. Presidents Council of Advisors on Science and Technology, Realizing the Full Potential of Govt-held Spectrum to Spur Economic Growth, July 2012.

Fehske, A, Gaeddert, J, and Reed, Jeffrey H. A new approach to signal classification using spectral correlation and neural networks. In *IEEE DySPAN*, 2005.

GNU Radio. GNU Radio. http://gnuradio.org/, 2017.

GNU Radio Build Script. GNU Radio Build Script. http://www.sbrac.org/files/build-gnuradio, 2016.

GNU Radio Companion. GNU Radio Companion. http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion, 2017.

Gunther, Jake, Gunther, Hyrum, and Moon, Todd. GPU Acceleration of DSP for Communication Receivers. *Proceedings of the GNU Radio Conference*, 2(1), 2017.

Hitefield, Seth and Clancy, T. Flowgraph Acceleration with GPUs: Analyzing the Benefits of Custom Buffers in GNU Radio. *Proceedings of the GNU Radio Conference*, 1(1), 2016.

Piscopo, Michael. Study on Implementing OpenCL in Common GNURadio Blocks. *Proceedings of the GNU Radio Conference*, 2(1), 2017.

Qt. Qt. https://www.qt.io/, 2017.

Roberts, Randy S, Brown, William A, and Loomis, Herschel H. Computationally efficient algorithms for cyclic spectral analysis. *IEEE Signal Processing Magazine*, 8 (2):38–49, 1991.

Rondeau, Thomas W., O'Shea, Timothy, and Goergen, Nathan. Inspecting GNU Radio Applications with Controlport and Performance Counters. In *Proceedings of the Second Workshop on Software Radio Implementation Forum*, SRIF '13, 2013.

VOLK. Vector Optimized Library of Kernels. http://libvolk.org/, 2017.