Array signal processing optimization in GNU Radio for tracking and receiving applications

E. Bieber[1], C. Campo[1,2], L. Bernard[1], H. Boeglen[2], S. Hengy[1], J.-M. Paillot[2]
[1] French-German research institute of Saint-Louis (ISL), Saint-Louis, France
[2] Université de Poitiers, XLIM, UMR 7252, Poitiers, France

Abstract

Among other missions the French German research Institute of Saint-Louis (ISL) works on array signal processing for secured communications between high speed projectiles and allied base stations. Within that framework, a projectile tracking receiving station based on commercial Software-Defined Radios (SDR) was developed using four channels to steer an antenna array and recombine the received signals, hence improving the gain of the receiving station. A transmitter embedded in the projectile sent data to the developed receiving station at a 2 Mbits/s. In order to decode and process in real time the data received by the four channel antenna array, a high sampling rate was required. As this highly resource consuming application resulted in sample overflows that is, in periodic losses of data between the SDR and the computer, an optimization of our algorithms computed on GNU Radio and the communication between our blocks proved to be necessary.

This paper intends to provide feedback on our optimization work. Some of the main problems we encountered and the solutions we propose to solve them are briefly exposed and will be further detailed in our oral presentation.

1 Introduction

Among other missions, the ISL works on developing secure communications between fired projectiles and ground stations for future smart ammunitions. Antenna arrays then offer many advantages such as directional radiation patterns that can be dynamically reconfigured to follow a moving transmitter, fight against hostile jammers or listeners, etc. In this context a SDR-based receiving station was developed using GNU Radio and the commercial Universal Software Radio Peripherals (USRPs) sold by National Instruments, and proved to be able to electronically follow a transmitter by steering a four element Uniform Linear Array (ULA), increasing the gain on the received signal. However in order to simultaneously decode the transmitted signal at a 2 Mbits/s baud rate, the sampling rate for all channels needed to be raised to 8 MSamples/s. To compose with the SDR requirements it was necessary for our laptop to receive data at a total rate of 33.33 MS/s (for all four channels), process the received data with our implemented algorithms such as beamforming and direction finding (DOA) that were introduced in [1], and record the whole in real time, resulting in a highly data consuming application. Our first attempt to run this application with a laptop equipped with an Intel i7 processor, 32 GB of RAM and a Samsung 850 evo SSD resulted in data overflows, i.e. in periodic data losses due to the lack of computation power, hence forcing us to think carefully about computation efficiency when implementing our application in GNU Radio.

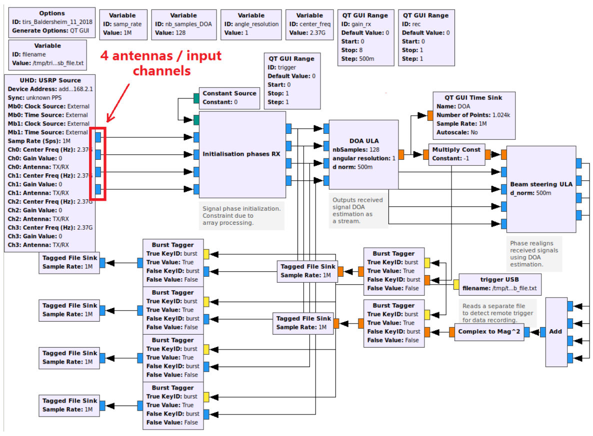

Figure 1: Flowgraph runnable at 1MS/s but creating data overflows at 8.33MS/s.

Fig. 1 exhibits a flowgraph that managed to perform projectile following at 1MS/s but created data overflows when higher sampling rates were required. The important number of streams and use of loops instead of vector oriented library kernels (Volk) were partly responsible for these overflows.

This paper does not focus on our application and results, but intends to present our work on code optimization, especially to extend the computation efficiency of our algorithms and flowgraphs developed in C++ in GNU Radio [2]. The remaining of this abstract briefly covers suggested improvements we have explored to avoid overflow issues.

2 Suggested improvements for optimization

The first and most obvious question that arises is the network throughput between the SDR and the laptop, as well as the laptop capability to record all the needed data fast enough. In the case of our application, the total 33.33MS/s sampling rate forced us to install a Thunderbolt 2 SANLink adapter. It can also be useful to create a RAMDisk if the drive is not capable to record fast enough. Reducing the amount of written data will have a positive effect on the bitrate: users should write binary files rather than ASCII ones and use data types with smaller memory size.

Once it is sure network throughput and data recording speed are not the bottlenecks that create data overflows, one can investigate his source code to enhance his application efficiency. Due to GNU Radio's way of processing streams as buffers of data, the *work()* method of a block is usually a two-level loop that parses each sample of each input stream. One should avoid multiple computations of invariant values inside loops, but also try to use optimized functions such as *memcpy()* or the Volk library [3] kernels instead of these loops whenever possible.

However even if correctly managing streams between blocks allows to spare resources, it remains important to limit those streams when they are expendable. Although it might be tempting as a fast implementation to simply add a stream to a block as a trigger or a way to share a variable between blocks, it is computationally expensive and can be responsible for data overflows when high sampling rates are required. In order to efficiently communicate information between blocks, GNU Radio natively offers the possibility to tag existing streams with metadata. Since a tag is associated to a sample of a data buffer, we can consider tags as a synchronous communication vector. For asynchronous communication GNU Radio allows blocks to send messages to other blocks. A message is a 1 to N communcation carried out by the sender: the receiver message handler is called for each pending message. As no native option is given for users to develop blocks that can asynchronously use a shared variable, we developed a new communication vector based on static variables that allow variables to be read and written by several blocks in a thread-safe way assured by a mutex. Further details on our proposed communication vector between blocks will be given in our presentation.

After information communication between blocks has been verified enabling real time scheduling option will allow GNU Radio threads to have priority over concurrent threads. Finally blocks using computationally heavy algorithms like DOA estimation, etc, can be bound to a dedicated processor core while less demanding threads are bound to a pool of remaining cores. It can be noted that sometimes splitting such blocks into several ones will take advantage of the multi-core environment.

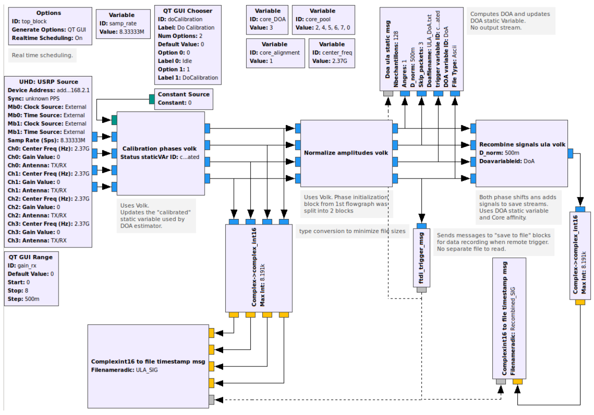

Figure 2: Flowgraph runnable at 8.33MS/s.

Fig. 2 shows an optimized version of the previous flowgraph that can be run on the same laptop at 8.33MS/s with a graphical view of the four received signals.

3 Conclusion

This paper presents the main modifications we brought to our developed blocks, making our application runnable for four channels at a 8.33MS/s sampling rate. An alternative communication vector between blocks that fits some of our particular needs has been mentioned, and all the suggested improvements presented above will be further detailed in our oral presentation.

References

[1] C. Campo, L. Bernard, H. Boeglen, S. Hengy, J.-M. Paillot, *Software-Defined Radio system for tracking application*, EuCAP London 2018.

[2] *GNU Radio 3.6.4.2 C++ API documentation* at `https://www.gnuradio.org/doc/doxygen-3.6.4/index.html`

[3] *Vector Optimized Library of Kernels (Volk) doxygen documentation* at `libvolk.org/doxygen/`