Distributed wideband software-defined radio receiver for heterogeneous systems

Master Thesis

Arnaud Durand

from

Fribourg

University of Fribourg

March 2015

Prof. Dr. Torsten Braun, Communication and Distributed Systems, UNIBE, Supervisor

Islam Alyafawi, Communication and Distributed Systems, UNIBE, Assistant

Foreword

The intended audience of this document is computer science students and engineers interested in distributed computing of signal processing systems or in an efficient implementation of a polyphase filterbank channelizer. No prior knowledge of distributed systems or digital signal processing is required. If the reader is unfamiliar with the GNU Radio framework, we do, however, recommend reading some official tutorials[1] as they will help the reader to understand the proposed implementation details.

Acknowledgments

I want to express my gratitude to Islam Alyafawi, initiator of the project. I am very thankful for his guidance, invaluable signal processing teaching and open-mindedness for new ideas.

[1] https://gnuradio.org/redmine/projects/gnuradio/wiki/Tutorials

Abstract

Recent years have seen an increasing need for computationally efficient implementation of software-defined radio (SDR) systems. Given the limitations of a typical SDR application running on a single machine, we present a distributed SDR system using high-performance techniques. To split a digital signal into multiple channels, we use an efficient digital signal processing technique: a channelizer. Distributed machines then process the channelized streams. To achieve this, we implement a load-balancer module and add new internet protocol communication capabilities to an existing SDR framework. Since the channelizer cannot be efficiently distributed, a single host machine has to process the entire wide-band signal. Furthermore, we optimize an existing implementation of a CPU channelizer and implement a new original GPU channelizer to push the system limitations further. The described techniques are applied to and tested with an existing implementation of a passive GSM receiver to add a full-band capability.

Table of content

List of tables and figures

List of appendixes

Acronyms

ADC	analog-to-digital converter
AVX	advanced vector instructions
DL	downlink
DSP	digital signal processing
FFT	fast Fourier transform
FIR	finite impulse response
FPGA	field-programmable gate array
GPU	graphics processing unit
GR	GNU Radio
GRC	GNU Radio Companion
GSM	Global System for Mobile Communications
MSPS	megasamples per second
PGM	Pragmatic General Multicast
SDR	software-defined radio
SIMD	single instruction multiple data
UHD	USRP hardware driver
UL	uplink
USRP	Universal Software Radio Peripheral
ZMTP	ZeroMQ Message Transport Protocol
ZMQ	ZeroMQ

1 Introduction

1.1 Context

With the ever growing popularity of wireless communications, the demand for highly innovative, flexible radio systems with a shorter time to market is increasing. This trend does not seem to slow down, but to accelerate as new types of devices, including smart objects and wearable devices, are becoming widely available. Current devices often support several technologies for wireless communications. The Global System for Mobile Communications (GSM), Long Term Evolution (LTE), WiFi, Bluetooth are just a few examples of them. The wide availability of these radio technologies enriches the ground for many add-on services. For fast development and testing, the Software Defined Radio (SDR) is used, which relies on software-based approaches without the hurdle of designing and building dedicated hardware. It is thus possible to design innovative systems at lower costs.

While a wideband SDR receiver offers many advantages in terms of costs and configurability, it sets several challenges to signal capturing. In particular, it requires a tremendous amount of computing power for signal processing as general-purpose processors (GPP) are not as efficient as specialized hardware. Depending on the system implementation, the computational complexity can grow linearly or exponentially as more bandwidth is processed. It is often not sustainable to run a full-band system on a single machine. Many systems, including GSM, uses frequency-division multiplexing (FDM) techniques, so the signal is divided into multiple channels. It is, therefore, possible to process these channels independently.

1.2 Project description

GNU Radio (GR) is a widely-used open-source SDR framework. It is a good candidate for SDR systems implementation thanks to its versatility and the availability of many user-contributed external modules. GR applications are implicitly parallel and thus scale very well with multicore CPUs. While we can upgrade the hardware to a high-end CPU with more cores for wide-band applications, this solution is not scalable as we would quickly hit another performance limit.

We propose a better scalable solution with a distributed system. By using available resources from different machines connected to the same the network, we can have access to potentially unlimited raw processing power. Any GR application having to process independent channels can benefit from our software. We will use an example application based on AirProbe, a GSM-sniffer. However, our research is not focused on GSM systems and can potentially be used in a wide range of applications, including radio astronomy and wide-band signal detection and analysis (WiFi, Bluetooth).

In this thesis, we will analyze how we can distribute a GR-based SDR system. We will also make the most out of available resources by applying optimization techniques to the critical sections, including the channelizer. The work will be validated on an existing, work-in-progress GSM receiver system [1]. This system captures GSM frames and store metadata for later processing, such as users identification and localization. It can do so for several channels in parallel, making it capable of wide-band analysis of several GSM channels.

1.3 Main goals

We want to distribute the workload of an SDR wide-band receiver. Only the SDR hardware receiver should limit an ideal system. If more processing power is required, we should be able to process more data by just increasing the available resources; a pool of processing machines.

To achieve this goal we need to fulfill several objectives:

- ❑ Distribute the radio signal over an IP network
- ❑ Create or use a load balancing system, so that processing is automatically allocated to one or more machines
- ❑ Remove or reduce system bottlenecks

1.4 Contributions

We propose and implement a solution to distribute the workload of a wideband SDR receiver using a load-balancer. We test and validate the proposed solution on the existing GSM receiver system so that each channel can be independently analyzed on different machines. Workload distribution presents many challenges:

- ❑ Channelize the signal on the host at a very high sampling rate in real time
- ❑ Assigning the channels to a set of worker machines according to available resources
- ❑ Make the best use of the limited link capacity
- ❑ Have a generic implementation suitable for a wide variety of projects

There are a few proposals for distributed SDR systems ([2], [3]). These proposals are however software architecture ideas supported by experiments on fixed scenarios. Furthermore, they did not release any tool. GR provides a set of communication tools for building distributed systems that can be used for a static infrastructure configuration. We have more advanced requirements from a communications standpoint, including load balancing and session data. We, therefore, need to come up with a new solution.

To channelize the signal at a very high sampling rate and in real time, we require an efficient implementation of a polyphase filterbank (PFB) channelizer: a component splitting a wide-band signal into equi-spaced channels. PFB channelizer implementations are widely available for various platforms ([4], [5], [6]). Due to the characteristics of the GR architecture, it may be difficult to adapt a code from another platform. GR includes a channelizer implementation for the CPU, which can take advantage of architecture specific SIMD (single instruction multiple data) operations. We will see that it is not always fast enough for our application.

To achieve our proposed system, we will:

- ❏ Further optimize the existing CPU PFB channelizer of GR to take advantage of fused multiply-add (FMA) operations
- ❏ Implement a new GPU PFB channelizer suited for GR
- ❏ Build a workload distribution system, including a load-balancer
- ❏ Create new tools for digital signal transport
- ❏ Test and validate our distributed system on a GSM receiver for message capturing

All of these features are provided as independent GR external modules: one module for each PFB channelizer implementation and one module for the distributed system set of tools. These modules are suited to any application having to analyze independent channels using the same SDR framework.

2 Related work

2.1 Software Defined Radio

An SDR system uses computer or embedded software to process radio signals. While a typical radio system uses dedicated hardware to process the signal in the analog domain, modern CPUs have become fast enough to handle the processing of sampled signals with relatively high data rate. This approach has many advantages: dynamic reconfiguration, overall flexibility, fast prototyping, and it does not require the skills and costs to create dedicated hardware. However, CPUs are nowhere as fast and efficient when compared to dedicated hardware for this type of processing. It, therefore, requires quite powerful hardware, especially to process high data rate signals in real time.

A multi-rate signal processing application typically consists of (1) an analog receiver - including analog filters, (2) a digital front end - placed right after the ADC and (3) a DSP application [7]. This is described in figure 1.

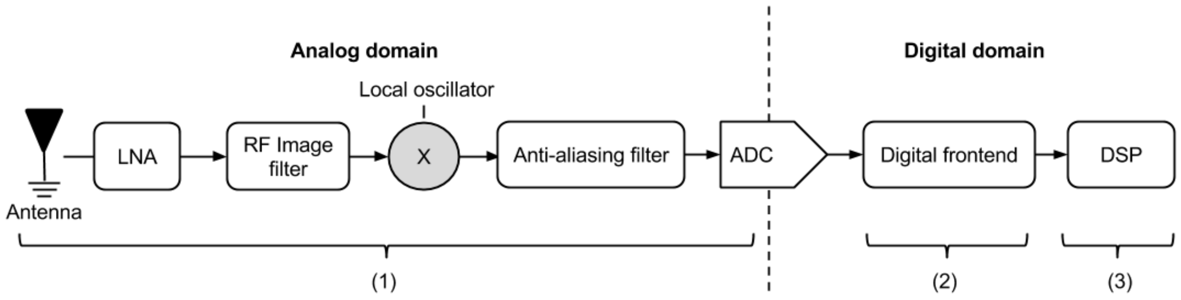


Figure 1 – Typical SDR receiver

2.2 GNU Radio

2.2.1 Introduction

GR is a framework to build SDR and signal processing systems. It was first published in 2001 and is a free software using the GNU General Public License (GPL). The project is written in C++ and Python. GR consists of several processing libraries and tools that can be used to build an SDR application. A GR application is built using flowgraphs. A typical GR flowgraph will contain some signal processing blocks, at least one source (e.g., a signal receiver) and at least one sink (e.g., a transmitter) connected together. The GNU Radio Runtime is the basic building block of all GR applications and provides the tools to create and execute a flowgraph. Developers have access to many modules, providing pre-built processing blocks for various applications. A module is an independent set of GR tools and libraries, and users can create their own, enabling reusable components to be easily shared within the community. GR also contains a set of tools, most importantly a GUI tool to create flowgraphs - the GNU Radio Companion (GRC), and a filter-designing tool.

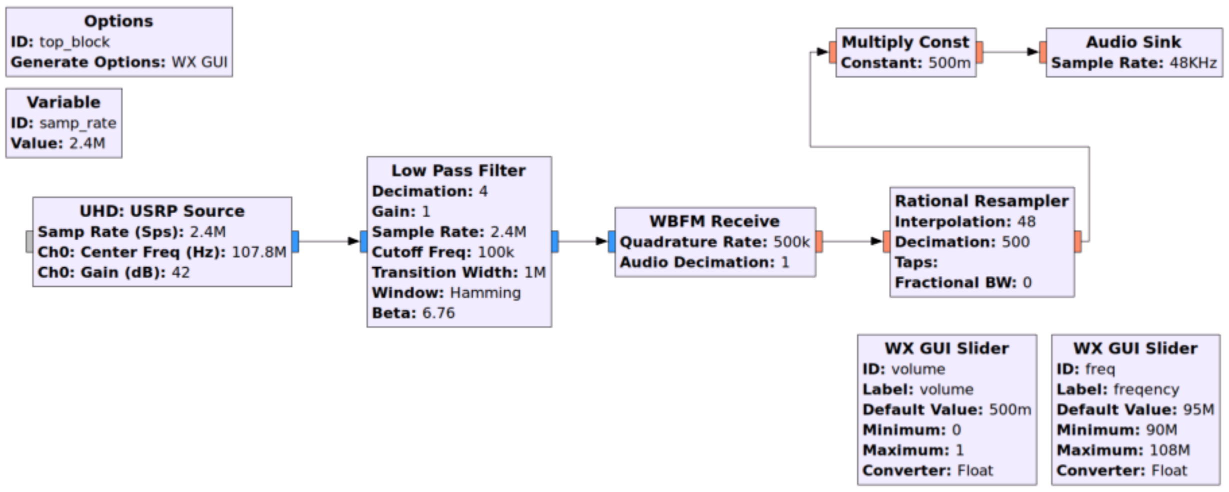


Figure 2 – FM receiver flowgraph made inside GRC

2.2.2 Architecture

GR is mainly built in C++, including the runtime and most processing blocks. GR Python applications are capable of using C++ modules using the SWIG[2] interface. SWIG is typically used to generate the 'glue code' to call C/C++ interfaces. Using this technique, Python can be used for high-level programming without any performance penalty. For this reason, Python applications are primarily used for flowgraph management or fast prototyping of blocks. We will not describe in details Python-specific features as we only used them for testing purpose.

There are different types of processing blocks, but they all share a set of common features; they consume and/or produce data and process it continuously inside a *work()* function. Blocks are connected through their input and output ports, and a block may have several input and output ports. A block type defines its relation to connected blocks. For example, a source block produces data but does not consume any, while a synchronous (sync block) has a fixed ratio between consumed and produced items. Block types include general, synchronous, source, sink, hierarchical and top block. Hierarchical and top blocks are special block types made out of other blocks. The top block is the main flowgraph, and there is always exactly one top block in a GR application.

Connected blocks exchange consumed/produced data inside buffers. Distributing the workload of a GR process is equivalent to running blocks in parallel. This process is done automatically when running a flowgraph: by default, each block is executed on its own thread. An application thus takes advantage of multicore CPUs without requiring extra work from the programmer. While thread priority of each block can be pre-configured for performance tuning, the thread scheduling is solely managed by the operating system. The GR scheduler is responsible for calling the *work()* function with optimal expected produced

[2] http://www.swig.org/

items, so there is no starving block in the flowgraph. The GNU Radio scheduler also handles buffer management so that there is no need for explicit synchronization inside processing blocks.

2.2.3 Communication mechanisms

Apart from the main data exchanged through buffers, GR provides several communication mechanisms for communication inside the flowgraph:

- ❑ **Messages**

GR messaging system provides asynchronous inter-blocks communication. It uses a dedicated message port. Messaging follows the publish/subscribe pattern.

- ❑ **Protocol data units**

Protocol data units (PDUs) is a convention to pass both data and metadata as polymorphic type (PMT) pairs. Polymorphic Types are "opaque data types that are designed as generic containers of data that can be safely passed around between blocks and threads in GNU Radio"[3]. So PDU is a specific PMT always encoded as <blob, dict>, with blob being the actual data and dict a dictionary of metadata.

- ❑ **Tagged streams**

Tagged stream is a parallel flow isosynchronous to the data stream. It contains PMTs of integers delimiting PDUs boundaries. Tagged streams are usually suited for handling packet-level protocols as it can be useful to separate packets, e.g. for calculating checksums. They can be used by regular blocks or by using special tagged stream blocks, which always consume data of exactly one PDU at a time.

- ❑ **Control ports**

Control ports is an interface for remote procedure calls (RPCs) that can be used for remote control, data visualization (plotting), debugging, etc. of GR blocks.

2.3 USRP N210 device

2.3.1 Overview

While SDR uses software to implement radio communication system components, we still need radio hardware to receive and/or transmit radio signals. The Universal Software Radio Peripheral (USRP) is a series of SDR-enabled devices developed by Ettus Research. For this project, we are specifically using a set of USRP N210. These devices can operate over a very wide spectrum range (50MHz – 6 GHz) and are, therefore, suitable for a wide range of applications. Note that other GR-compatible SDR devices are available from various vendors and these could be used as well.

[3] http://gnuradio.org/doc/doxygen/page_pmt.html

Figure 3 – USRP N210 - Ettus Research™

The device itself consists of an analog receiver (the daughterboard), a Xilinx Spartan 3A FPGA and a Gigabit Media Independent Interface (GMII). The architecture is detailed in figure 4.

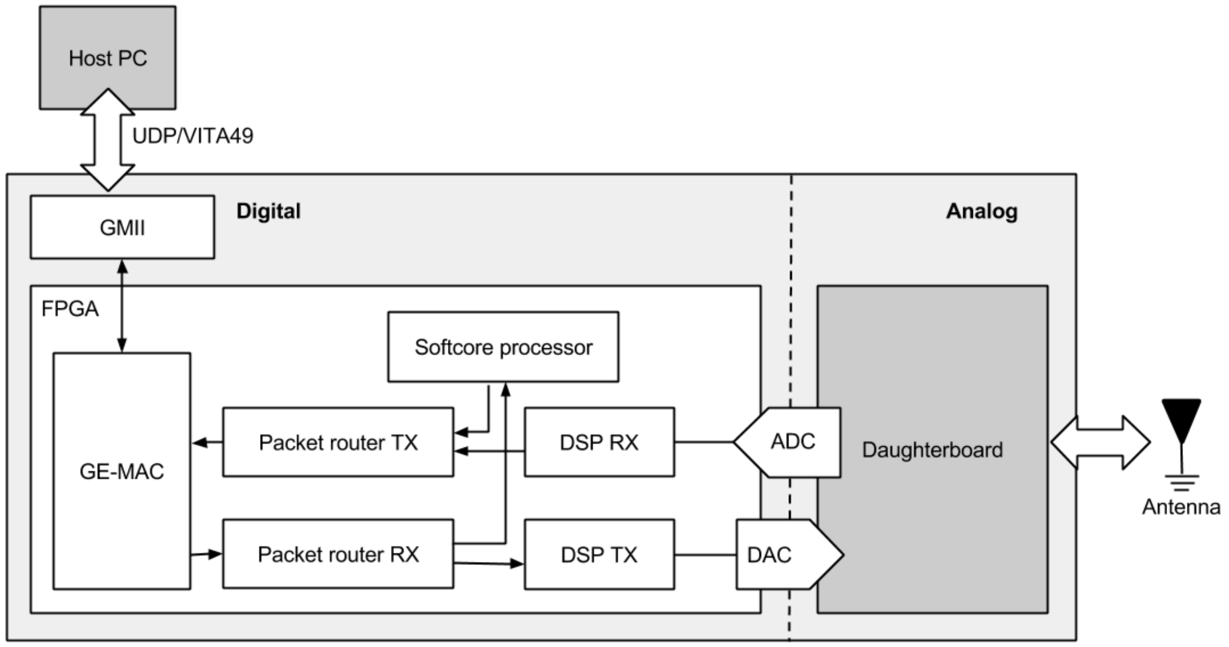


Figure 4 – USRP N210 architecture

The USRP N210 can be configured to capture 8 or 16 bits precision in-phase, quadratic (I/Q) samples. Data from an I/Q sample is made of two independent consecutive samples. The actual ADC precision is 14 bits. The FPGA reduces the data to 16 bit samples (8 bit I and 8 bit Q) when configured in 8 bit mode or uses 32 bits per sample otherwise (16 bit I and 16 bit Q). The N210 is internally able to sample at a rate up to 100Msps, but only the FPGA can process samples at this speed. When using the default FPGA application, the signal is decimated and sent to the UHD driver through the Gigabit Ethernet interface. According to the documentation, the N210 is able to capture up to 25 Msps in 16 bits mode[4]. This has been validated using a GR-bundled sample application. At this rate, we measured an average of 816.85 Mbit/s from the USRP to the host Ethernet interface using the vnstat[5] tool.

USRP N210 specification excerpt (from datasheet[6])	
FPGA	Spartan 3A-DSP 3400
Interface to host	Gigabit Ethernet
ADC Sample Rate	100 MSPS
ADC Resolution	14 bits
Sample Rate from Host (8b/16b)	50/25 MSPS

Table 5 – USRP N210 specification excerpt, Ettus Research™

The USRP communicates with applications and drivers using the VITA protocol, a standardized SDR protocol to provide interoperability between various components. More precisely, it uses VITA49 as encapsulation above the UDP layer. A VITA49 packet is described in figure 6. As we can see, there is very few data overhead due to the verbosity, enabling the USRP to output 25 MSPS while staying under 1 Gbps.

Bit 31 . . . . Bit 0
Header (1 Word, Mandatory)
Stream Identifier (1 Word, Optional)
Class Identifier (2 Words, Optional)
Integer-seconds Timestamp (1 Word, Optional)
Fractional-seconds Timestamp (2 Words, Optional)
Data Payload (Variable, Mandatory)
Trailer (1 Word, Optional)

Figure 6 – VITA protocol structure, Pentek, Inc[7]

[4] http://www.ettus.com/content/files/kb/application_note_selecting_a_usrp.pdf
[5] http://humdi.net/vnstat/
[6] http://www.ettus.com/content/files/07495_Ettus_N200-210_DS_Flyer_HR_1.pdf
[7] http://www.pentek.com/tutorials/17_2/VITA.cfm

2.3.2 UHD driver

The UHD driver is an open-source software component providing access to the USRP from the host platform. The USRP can be used through the UHD API or, more likely, through the connectors provided for various platforms. The provided connectors for GR comprises both USRP sink and source blocks, respectively for receiving and transmitting signals through the USRP. The UHD provides access to multiple USRP devices in a homogeneous setup. It means that applications using UHD can benefit from multiple USRP capturing data at different locations. Up to our knowledge, there is no built-in mechanism to use a single USRP from several hosts simultaneously using the UHD driver.

2.4 Running example: a passive GSM receiver

Thorough this document, we will test and use the presented techniques with an existing, work-in-progress passive GSM signal capturing project. The general idea is to detect GSM messages and to store metadata on a database for later processing. We will experiment and use the frequencies of the E-GSM-900 frequency band and will, therefore mention it simply as GSM. GSM uses a time division multiple access (TDMA) channel access method, so each user is allocated to a particular channel and timeslot. The system is able to capture messages on both directions: downlink (DL) and uplink (UL). It takes advantage of the fact that GSM downlink (DL) and uplink (UL) are synchronized: channels and timeslots that carry the data for a user are the same for DL and UL, but shifted by 45Mhz. By scanning the DL band, it is possible to get a map of allocated channels and time slots of nearby cell towers. Scanning GSM DL opposes few technical issues as we can capture frames with a constant power and synchronization delay. However, capturing messages on UL is not an easy task as mobile devices adjust their delay and output power according to their location with the serving base station.

The passive receiver system is based on GNU Radio for digital frontend and DSP and some GSM frame detection and capturing is based on AirProbe, an open-source GSM sniffer project. Figure 7 shows the different components of the system [1]. The GR flowgraph consists of a UHD source, a digital front-end where we split the signal into M GSM channels, a few signal processing modules for signal adjustment, and some passive receiver modules for GSM signal detection. Note that all modules can run in parallel, but the digital front-end requires synchronization.

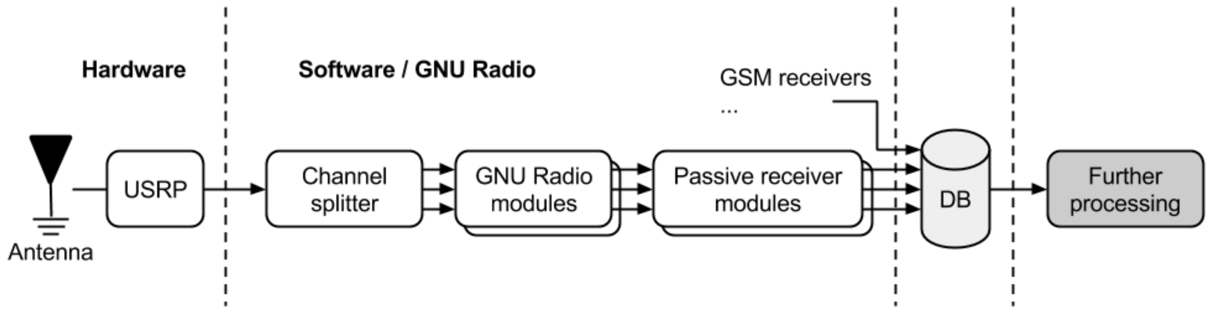


Figure 7 – Passive GSM receiver, Islam Alyafawi

3 Distributed wide-band receiver

3.1 Distributed architecture approaches

3.1.1 Introduction

We described an example GSM receiver system. We also stated that this system is computationally intensive, to the point where we can only listen to a few channel when using a high-end consumer desktop machine. Naturally, we want to optimize our system and distribute the workload on other machines to be able to listen to more channels. Ideally, we want a system that could either use (1) the whole USRP bandwidth or (2) the whole capacity of machines (workers) at our disposal. These workers determine our total processing power. If we are in this ideal situation, then we can claim that the system is scalable. In the GSM system, this is the signal processing blocks as well as the GSM signal detection, which require a lot of processing power. It corresponds to the module implemented in GR (highlighted in figure 8).

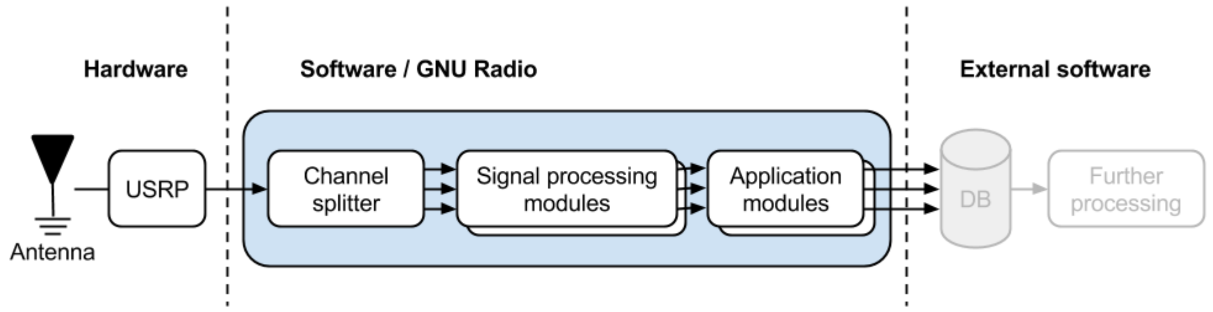


Figure 8 – Example SDR receiver application

In the next section, we will describe the possible approaches to distribute the workload of such a system.

3.1.2 Raw signal multicast

A naive approach would be to send the unfiltered signal to every worker. It is equivalent to broadcasting/multicasting the data coming out of the USRP to every worker. Signal filtering requires a decent amount of computing power on the host machine (more on this in chapter 4). It is a hard constraint: if we do not have enough computing power on the host, no matter how many workers or how powerful they are, the system will not be able to process more channels that the host machine can filter. Using multicast approach, we push all signal processing logic to the workers (as described in figure 9).

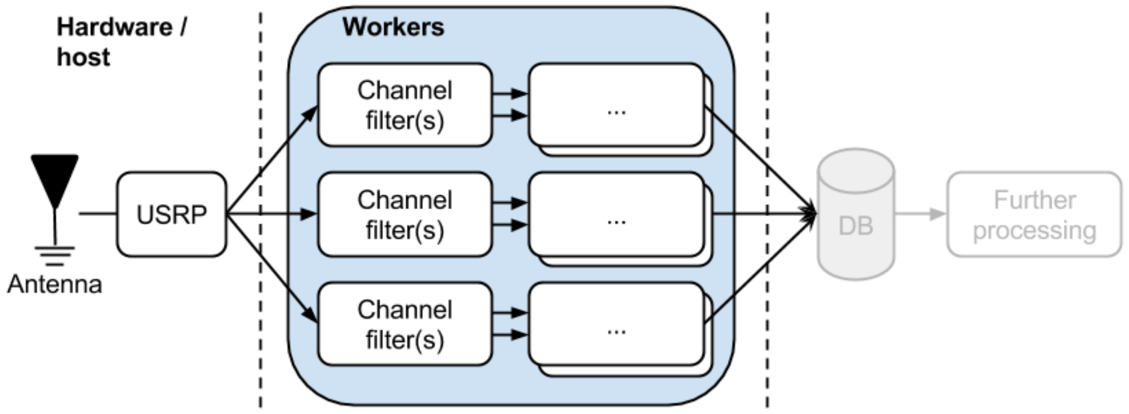


Figure 9 – Distributed SDR receiver using multicast

Indeed, using TCP to distribute the whole signal to many workers would be unsustainable, as it would require to maintain a very high data rate connection with each worker. For example, for a sampling rate of 25 Msps, each connection would require a data rate roughly equal to what we measured out of the USRP, which is 816.85 Mb/s. To overcome this problem, we take the advantage of network multicast. To do multicast, we are limited to connectionless protocols like UDP or Pragmatic General Multicast (PGM). While UDP is very common and does not require presentation, PGM is a transport layer protocol dedicated to multicast. Compared to UDP, PGM is reliable due to its use of negative acknowledgments (NAKs). It is available on most operating systems yet only supported by some network devices. If we are stuck with UDP but need the reliability ability, it is still possible to implement it in the application protocol. Unfortunately, there is no congestion control at the protocol level using PGM or UDP. It means that the link capacity should be either evaluated in advance or continuously in the application layer. Depending on the network usage and the physical topology we may require dedicated links as the whole signal would have to reach each endpoint. In opposition to a channel unicast approach, it is not possible to drop channels in case of congestion. Dropping samples will severely alter the application functionality, and this situation is not suitable for real-time support.

Using this approach, the network is required to handle the (re)transmission of data to every worker at a high data rate. Enabling high throughput multicast requires specific hardware features and configurations. Storm control is a router and switch feature for limiting broadcast traffic rate and preventing packet flooding. It is often enabled on customer devices, but most of the time not configurable. Enterprise class hardware usually enables this parameter to be configured on a per-port basis. As we require very high throughput for our application, it is required to disable storm control to support signal broadcasting. IGMP snooping feature is also desired if using the IPv4 addressing scheme. In a multicast context, this allows routers and switches to distribute the multicast streams selectively to recipients rather than broadcasting packets to every port. Such a situation would create much unwanted traffic for undedicated networks.

While having some specific network infrastructure requirements, signal multicast may seem appealing:

- ❑ No computational bottleneck before task splitting
- ❑ Requires very few software modifications

Assuming that we are implementing this approach, each worker would have to do the filtering process of its own channels out of the whole signal. It is a very intensive process, and we cannot use efficient filtering techniques described in section 4.1. That means that we overall require more processing power and may end up with situations where workers would not be able to filter their channels for a high sampling rate.

3.1.3 Channel unicast

The other approach we might use consists in doing all the filtering process on a single machine. Using this approach, the USRP is connected to a host machine which takes care of all the filtering process. Filtered channel signals are then sent to the appropriate workers. According to the Nyquist-Shannon sampling theorem, when filtered to a particular band (using lowpass or bandpass filter), the signal can be decimated until a certain threshold without introducing aliasing [8]. In our case, we can decimate each channel signal to $\frac{1}{M_{channels}}$ of the initial sampling rate. Compared to the previous approach where we needed to transmit overall $(rate_{sampling} \times size_{sample})^{N_{workers}}$, here we only need to transmit $\frac{rate_{sampling}}{M_{channels}} \times size_{sample}$ per channel, which is $rate_{sampling} \times size_{sample}$ overall.

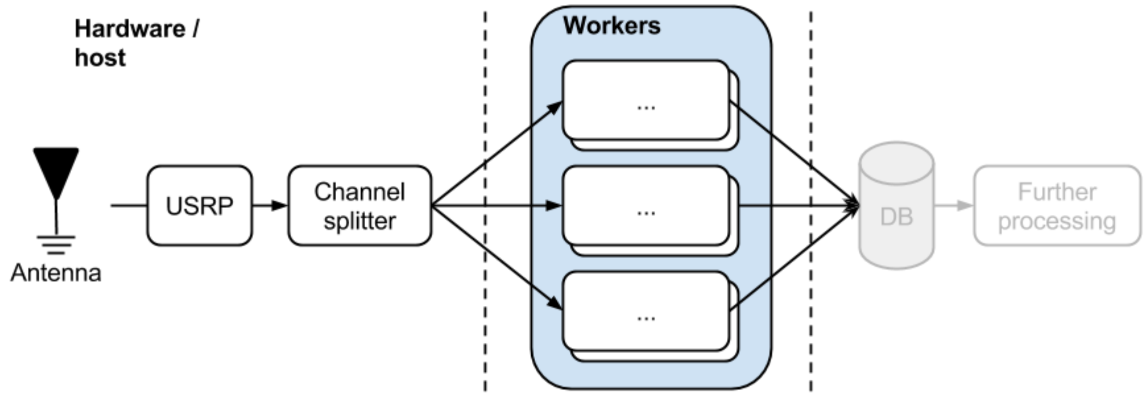


Figure 10 – Distributed SDR receiver using unicast

Using this approach, the host machine needs to be fast enough to process the filtering of its channels. Otherwise, the UHD buffer will overflow and samples will be dropped. It can be problematic due to the high complexity of channel filtering. We will address this issue in chapter 4. If an application requires processing a very large band which a single host is not able to handle, we have to use two or more host machines with their own SDR receiver. By using different center frequencies at a reduced sampling rate, we can reduce the processing requirements of the hosts.

We judged the channel unicast as the best approach, as it is computationally much more efficient, and it does not require specific network infrastructure.

3.1.4 Hybrid approach

A hybrid approach would also be possible. In this case, we would have to use multi-stage filtering. However, it would not benefit from the efficient PFB channelizer technique as it requires intermediate synchronizations. We also judged that the benefits would not overcome the complexity of implementation and infrastructure management.

3.2 Load balancing

3.2.1 Introduction

As we estimated that the channel unicast was the best approach, we, therefore, need to distribute the workload. Ideally, we want to have some dynamic behavior, not a fixed preconfigured set of workers. If we outline the situation, we have one host machine filtering a signal into channels which are sent to workers to be processed. A worker can process one or several channels, depending on its capacity and its workload. There can also potentially be more than one host machine as we might want to capture more channels and/or capture channels from both downlink and uplink. This precise situation requires load balancing mechanisms.

We chose to use a token-based approach: our workers send a token when they are ready to process data from more channels (figure 11). Workers send as many tokens as they can process channels. They also resend tokens upon disconnections to signal a new task can be allocated to them. To implement this behavior, we require a signaling mechanism as well as stable points on the network who can attribute tokens to host machines. We will use the ZeroMQ messaging library for the signaling mechanism and a broker to manage the tokens, as described in the next sections.

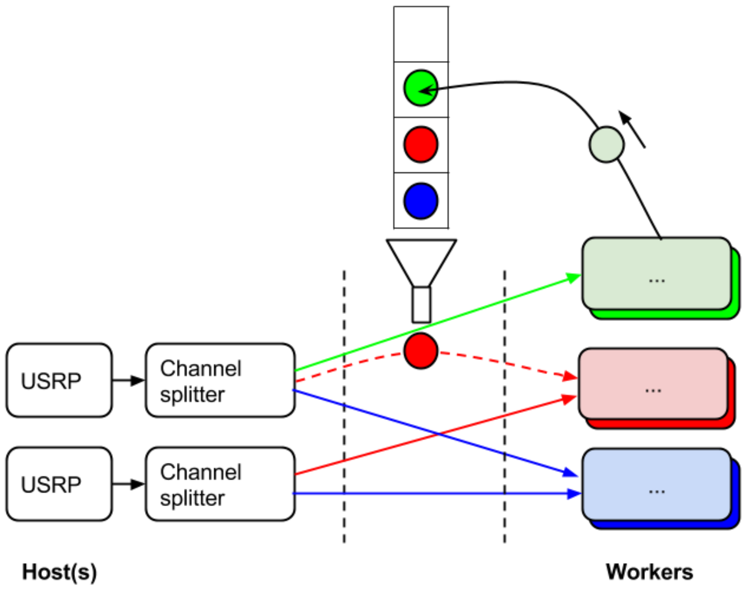


Figure 11 – Load balancer token mechanism

3.2.2 ZeroMQ (ØMQ)

ZeroMQ (ZMQ) is a general purpose message library available for many programming languages [9]. It can be used for intra-process communications (as a synchronization mechanism), inter-process communications (IPC) or network communications. Compared to competitors (ActiveMQ, RabbitMQ, etc.), ZMQ targets performance and simplicity. It does not provide any built-in persistence mechanisms: when the broker is down, queues and items are lost. According to benchmarks[8], ZMQ is faster than its competitors, sometimes by several order of magnitude. It is therefore more suited for very intensive messaging or synchronization systems.

ZeroMQ provides several types of sockets which abstract the transport protocol: they might be arbitrarily used with TCP, PGM or ePGM (PGM over UDP) for network communications without any impact on the code.

Our interest in ZMQ grows due to an experimental GR module, gr-zmq. While this module does not provide any useful features for our application, we will take advantage of the ZMQ library to build a load-balancing mechanism. We also believe that any module proposal is more likely to be accepted by the GR community if having the same dependencies.

3.2.2.1 The ZeroMQ Message Transport Protocol

When ZMQ is used on par with a connected transport layer protocol such as TCP, it uses the ZeroMQ Message Transport Protocol (ZMTP)[9] to exchange messages between peers. While it is not necessary to understand the underlying protocol when using the ZMQ library, we will highlight a few key aspects of this protocol that will help understanding the techniques described in the next sections.

One important feature is related to framing. The TCP protocol carries streams of octets with no delimiters. ZMTP adds delimiters on top of the stream and the content between two consecutive delimiters is called a frame. Note that the frames we are describing here are specific to ZMQ and have nothing to do with TCP frames. Frames content could be either ZMTP commands or user-defined messages. Messages consist of one or several frames and are always sent atomically, no matter how many frames they encapsulate. One key aspect of ZMQ is that message framing can be used as delimiters, meaning that we can split a message in the application layer.

Another important aspect is that the protocol makes no distinction between clients and servers unless when using secure authentication. Any participant of a ZMQ exchange will, therefore, be referred to as peer.

3.2.2.2 ZeroMQ sockets

BSD socket (the traditional socket) API allows applications to use network sockets. ZMQ sockets are similar in concept but are linked to specific communication patterns. These communication patterns

[8] http://blog.x-aeon.com/2013/04/10/a-quick-message-queue-benchmark-activemq-rabbitmq-hornetq-qpid-apollo/
[9] http://rfc.zeromq.org/spec:37

enable better abstraction on the underlying network communication and topology. We will describe them in the next paragraph. ZMQ sockets also provide additional features like message framing, multiple endpoints and background I/O. ZMQ sockets are portable, so they do not depend on any operating system-specific set of features.

ZMQ provides several types of sockets used for different scenarios:

- ❑ REQ/REP: socket following the request-reply pattern
- ❑ ROUTER/DEALER: variation of REQ/REP
- ❑ PUB/SUB: socket following the publish-subscribe pattern
- ❑ PUSH/PULL: socket following the pipeline pattern

The request/reply pattern is the most basic pattern. It is similar to the client/server model. A communication following this pattern is always alternating with request and reply. A REQ socket will always block until it gets a response, and a REP socket will always block until it gets a request. For this reason, it is not possible to make two consecutive requests without receiving a first reply back. The ROUTER/DEALER sockets behave almost similarly to REQ/REP except: (quoted from [10])

- ❑ The REQ socket always sends an empty delimiter frame before any data frames; the DEALER does not.
- ❑ The REQ socket will send only one message before it receives a reply; the DEALER is fully asynchronous.

We can guess from these differences that we can emulate REQ/REP sockets by adding an empty delimiter frame and observing a strict alternation between requests and replies. It enables REQ/REP and ROUTER/DEALER sockets to talk together.

The publish-subscribe is a classic pattern where the publishers send messages to peers who subscribed to messages/topic (the subscribers). Using this pattern, the publisher has no knowledge of subscriber's identity and how many are they. The pipeline socket enables automatic load-balancing across a set of consumers. The producer(s) sends messages through the push socket to consumers, but only one consumer gets each message through the pull socket. We will not describe these two patterns further as we do not require them. ZMQ patterns are documented extensively in the official ZMQ documentation [9].

3.2.2.3 Devices

ZMQ devices are stable points on the network. As ZMQ does not use the client/server model, it requires stable points on the network to enable dynamic topologies. A ZMQ device is a piece of software providing a bootstrap for peers, enabling them to communicate with other peers without prior knowledge. As opposed to some other messaging libraries, devices are not ready-to-use executables.

[10] http://zguide.zeromq.org/php:chapter3#ROUTER-Broker-and-DEALER-Workers

Instead, they are either implemented by the developer using ZMQ sockets or configured from the library templates. There are several device types according to the sockets they use:

- ❑ Queue: uses REQ/REP sockets
- ❑ Forwarder: uses PUB/SUB sockets
- ❑ Streamer: uses PUSH/PULL sockets

We can also imagine other devices with more complex configurations.

The queue device is used with clients/servers configurations (see figure 12). When a client emits a request to the device, it is forwarded to the servers. The device also relays the response to the original client. We will use a slightly modified version of the queue device for the load balancer.

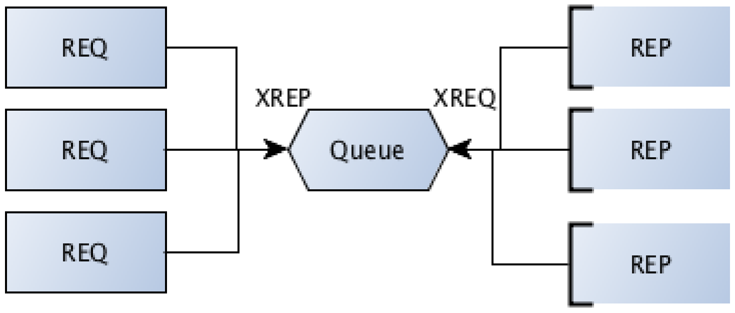


Figure 12 – ZMQ queue device, pyzmq documentation[11]

3.2.3 Roles

We already highlighted few roles in our load balancing system - clients (hosts), workers and a broker managing tokens. Let's define them more precisely:

The broker does the following:

- ❑ Accepts connections from a set of clients.
- ❑ Accepts connections from a set of workers.
- ❑ Accepts requests from clients and holds these in a single queue.
- ❑ Sends these requests to workers using the load balancing pattern.
- ❑ Receives status signaling from workers.
- ❑ Receives replies back from workers.
- ❑ Sends these replies back to the original requesting client.

The client does the following:

- ❑ Sends requests to workers.

[11] http://learning-0mq-with-pyzmq.readthedocs.org/en/latest/pyzmq/devices/queue.html

- ❏ Receives replies back from workers.

The worker does the following:

- ❏ Sends signaling status to the broker.
- ❏ Receives request from clients.
- ❏ Sends replies back to the original requesting client.

3.2.4 Signaling pattern

The signaling pattern between the client (e.g., USRP source) and the worker is a modified version we propose to the load-balancer[12] from ZMQ documentation. The broker device is always routing messages to an available worker using a single queue of tokens.

In figure 13, we present a simplified scenario with one client and one worker.

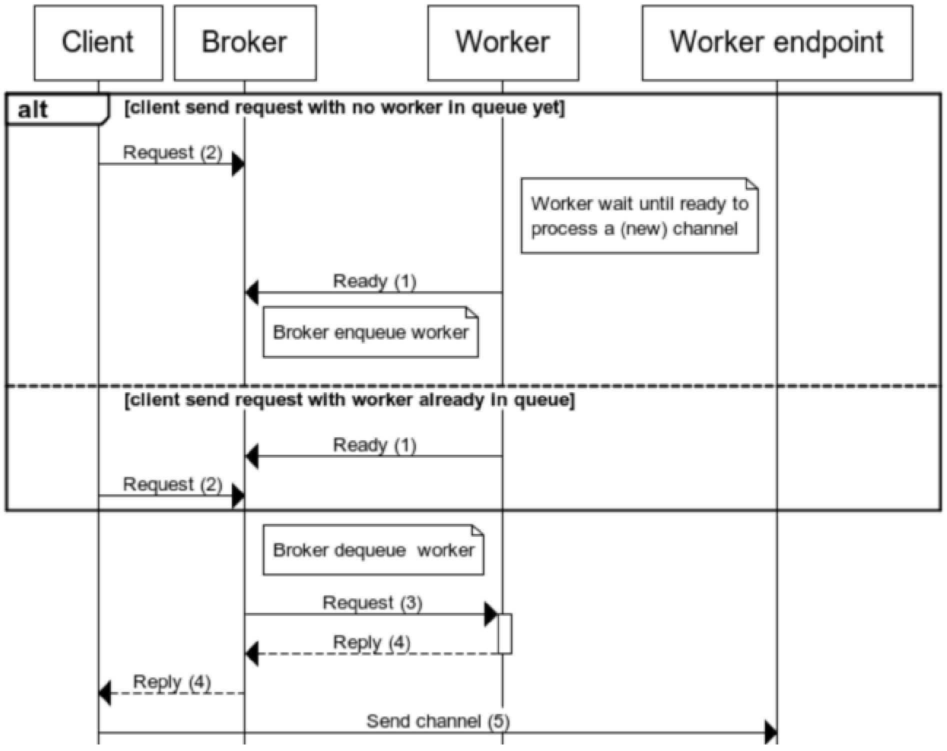


Figure 13 – Load balancer sequence diagram

The worker sends a signaling message (1) to the broker to indicate that it is ready to accept processing a new channel. The message is then added to a queue by the broker. When a client sends a request (2) containing session data (frequency, etc...) to the broker to ask for a worker to handle a channel, a worker is dequeued, and the message is sent to the worker (3). Alternatively, the client may send the request (2) before any worker is ready and wait for the worker to send the signaling message (1). The worker stores

[12] http://zguide.zeromq.org/php:chapter3#A-Load-Balancing-Message-Broker

the session data and replies (4) to the client its hostname and listening port. The client is then ready to connect (5) to the worker TCP endpoint.

3.2.5 Broker

The broker is a load balancer which distributes tokens. It is a stable point on the network (see figure 14), which is a device in ZMQ terminology. It is an application written in C++ inspired by the proposed ZMQ load balancer implementation. As it is a completely independent application, it could as well have been implemented in other languages thanks to the interoperability of ZMTP. In fact, we implemented a testing prototype in Python first to validate the design.

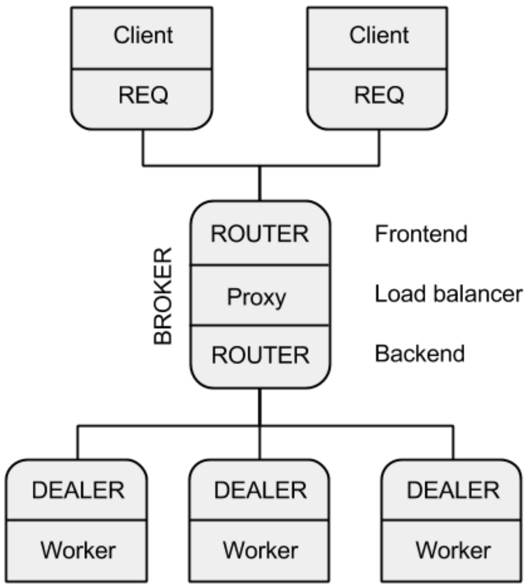


Figure 14 – Load balancer broker, adapted from zguide[13]

The broker consists of two ROUTER sockets and a proxy. The proxy itself is responsible for the load balancing. It contains a single token queue and follows this simple algorithm:

```
Init()
    token_queue <- []
Do forever:
    message <- backend.Poll()
    if message = 'READY':
        token_queue.Enqueue(message->sender)
    else: # message is a reply to client request
        frontend.Send(message->recipient, message)
    if tokens >= 1:
        request <- frontend.Poll()
        if request: # do we get a request?
            worker = token_queue.Dequeue()
            backend.Send(worker, request) # forward request to a worker
```

Listing 2 – PFB channelizer pseudocode

[13] http://zguide.zeromq.org/php:chapter3#A-Load-Balancing-Message-Broker

If we compare our broker to the load-balancer proposed in the ZMQ documentation, there is a fundamental difference: in the proposed ZMQ algorithm, the worker uses a pure request-reply pattern with the broker. It uses a Least Recently Used (LRU) queue to select a worker. It works because a worker is automatically reset as available by the broker when it replies to a job request. This mechanism applies only to jobs limited in time: upon task completion we take a new one. However, our system is quite different as tasks are channel processing, which are not limited in time. We can fix this by turning the LRU-based queue into a token mechanism: requests do not represent jobs anymore, but session tokens. It requires turning the worker socket into a DEALER socket as the worker does not follow a strict request-reply pattern anymore. Instead, it can emit to the broker as many tokens as channels it can handle.

We can already deduce that there are two possible behaviors for the worker:

- ❑ the worker emits all the tokens in advance
- ❑ the worker creates tokens one by one

If a worker know in advance how many channels it can handle, it can emit a fixed amount of tokens. It requires some benchmarking of each possible hardware combination we might use to run the worker application. It also requires a minimum of resources to be available at all times. It is especially an issue if we are not using dedicated machines: the resources allocated to our process or virtual machine might greatly vary over time.

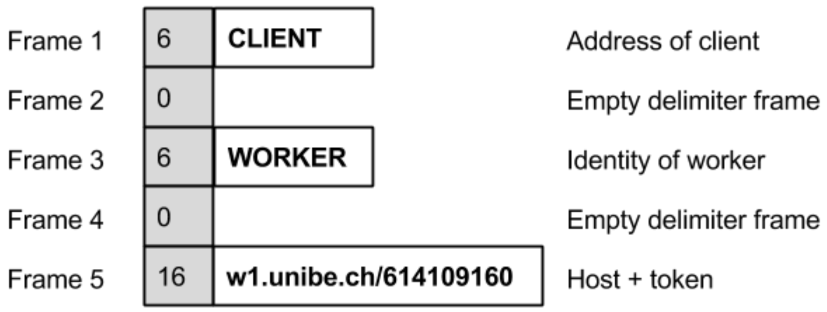


Figure 15 – Token message received by DEALER backend socket

If the workers emit its tokens one by one, it is possible to adapt the token creation according to the machine workload or by using a combination of rules. In other words, this is a situation where workers adopt some “reactive” behavior using overload control.

In section 3.3, we present more details about how clients and workers are interacting with the broker and how we use tokens to allocate new sessions.

3.3 GNU Radio Network modules

3.3.1 Radio signal traffic

Until now, we talked about distributed architectures and workload distribution. However, we did not define precisely the radio data we are carrying over the network. As for most SDR applications, we are processing I/Q samples. These are captured by the USRP receiver, transferred to the host and are then flowing through the buffers of the GR graph. While the USRP N210 datasheet explicitly states that the ADC can sample in 14-bit precision, we already stated that these are 16-bit precisions numbers (integers) inside the FPGA. Once arriving at the UHD driver source, these are converted to IEEE 754 floating point numbers. It presents the consequent advantage to be supported natively by all platforms and to be computed using floating-point units (FPUs). I/Q samples are complex data. These are typed gr_complex in GR, but are in fact only a redefinition of the C/C++ std::complex. Complex numbers are represented as two consecutive single-precision floating point numbers, thus taking 64 bits.

Now that we know the type of data we will be carrying, another important question remains unanswered: what is the data rate? It directly depends on the receiving sampling rate. We stated that the USRP can support a sampling rate up to 25 MSPS. Ideally, our sampling rate should always be equal or superior to the Nyquist rate[14], as we do not want to introduce aliasing in the system. However, the GSM receiver's modules need a precise data rate for each GSM channel: $rate_{GSM} = \frac{1625000}{6} \cong 271 \times 10^3 \; SPS$, which is equivalent to the data rate of the GSM signal itself. From these numbers, we can calculate a few elements:

- ❑ we need to sample at $rate_{GSM} \times M_{channels}$
- ❑ we can capture up to $\frac{25\times10^6}{rate_{GSM}} = 92 \; channels$ using the USRP N210

3.3.2 TCP blocks

The most important part of workload distribution of real-time signal processing software is how to distribute the signal itself. GNU Radio provides several blocks to distribute an application. UDP source and sinks provide unreliable connectionless communication and are available as a C++ blocks. TCP source and sinks provide reliable communication and are available as Python blocks. These Python modules are simple wrappers of the Python socket module. There is also the experimental gr-zmq module we previously mentioned, but it is suited for packet-based (on the GR level) transmissions and not for high throughput streams. Our application requires reliable transmissions, sessions as well as congestion control, so TCP is the natural fit. Unfortunately for us, the API does not provide much control over the sockets, and we cannot use these modules from C++ applications that do not provide SWIG interfaces. Note that the C++ blocks can be used from Python using SWIG (given an appropriate interface is provided), but the inverse is not possible. We thus decided to implement our TCP networking blocks that will give us the opportunity to be extended to fit closely to our requirements.

[14] http://en.wikipedia.org/wiki/Nyquist_rate

For our networking blocks, we decided to strictly observe the original API design of the GR TCP (Python) block so that it can be used on its own as a full-featured alternative. These are standard interfaces providing basic connection control like *connect()* and *disconnect()*. Here the TCP source (tcp_source block) is acting as a server, and the tcp_sink is a client. Workers and clients respectively use them. We also required some more features to use our load balancing techniques: the tcp_source *bind()* is used to bind on an existing socket, enabling us to listen for worker requests on only one port. We also added some notification mechanisms so the client/host is aware of the disconnection and can create/get a new token.

On a technical side-note, we are implementing these blocks using the Boost.Asio[15] library. The Boost library is a general purpose C++ library similar to the C++ standard library but with many more features. GR already heavily use this library, so we decided to use it to avoid external dependencies and with regards to the original GR design. While the tcp_sink could use either synchronous or asynchronous operations to send data inside the *work()* function, the tcp_source needs some asynchronous mechanisms: it acts as a server and the *work()* function is only called periodically by the GR scheduler. So we need a continuously running IO service (a thread), which avoids the socket buffer being full. This service continuously fills an intermediate buffer, which is then copied to the output buffer when the work function is called. To implement the notification mechanisms, we used the callback technique. While we usually use the observer pattern in this situation, we can take advantage of the Boost bind[16] mechanism to have a loosely coupled code with an optional callback.

3.3.3 Client

The client manages the transition between the host machine and the workers. What we want is to distribute as many channels as possible among workers. Therefore, its role is to obtain as many tokens as it can from the broker. Using sessions information's returned from a worker, it then connects the TCP sinks to the appropriate worker endpoint. The relationship of the client with system components is described in figure 16. Workers also get new tokens upon disconnections to connect to a new worker.

[15] http://www.boost.org/doc/libs/1_57_0/doc/html/boost_asio.html
[16] http://www.boost.org/doc/libs/1_57_0/libs/bind/bind.html

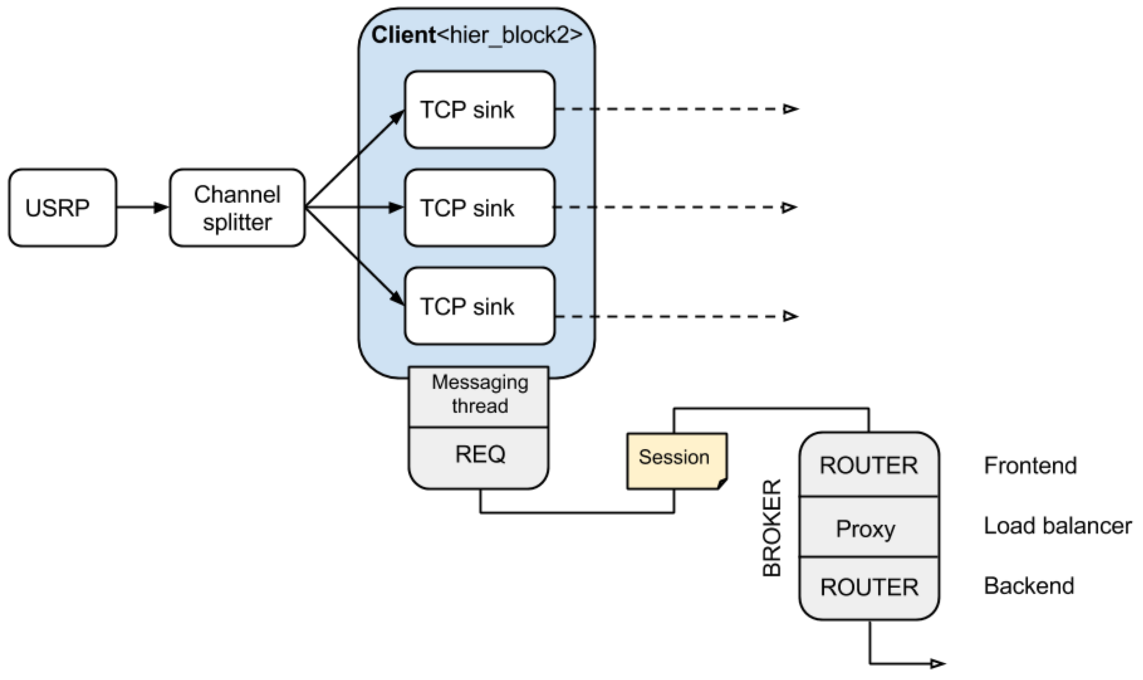


Figure 16 – Client application overview

The client is a GR block of type hier_block2, which is a hierarchical block. It contains the TCP sinks which are directly connected to its input ports. It means that the client can be used like any other blocks as the application and its source (the channel splitter here) never interact with it directly. This is a nice property as it ensures our system is generic enough to be used as general purpose load balancer for any GR applications. The client is simply instantiated with the number of input ports (channels) and router frontend hostname.

3.3.4 Worker

The worker role is a bit more complex. Not only it has to distribute the tokens, but it also has to manage a set of sessions. In the GSM receiver, the session token is linked with session data: we want to store data (e.g. the channel center frequency) about the processed channels to use it in the receiver configuration. When a client makes a request to the broker, this request also contains session data that is stored by the worker upon arrival. Interactions of the worker with system components are described in figure 17.

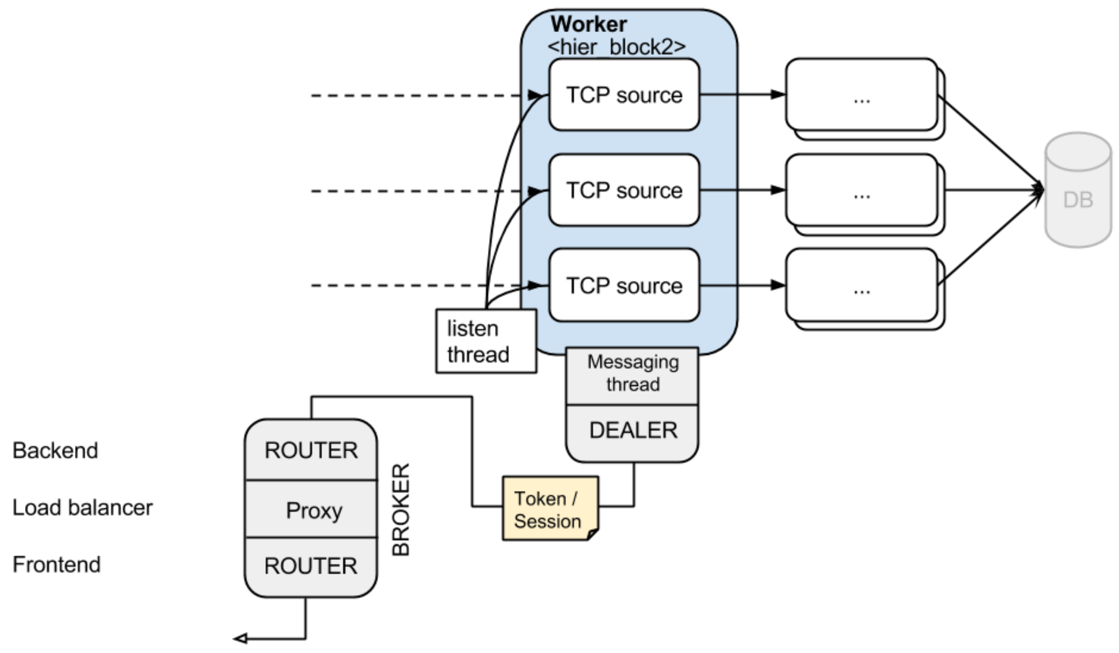


Figure 17 – Worker application overview

We also want our worker to have only one listening port, so we do not need a whole range of listening ports to be available and configured. Once a client gets a response from a worker, instead of connecting directly to a TCP source block, it connects to an endpoint running on a dedicated thread (listen thread of figure 16). The stream always begins with a 32-bit number in TCP/IP network byte order which is a token identifier. Using a map of previously stored sessions, we can bind the socket with the appropriate TCP source which was already pre-configured with the session data.

Indeed, we need to have some interactions with the application top_block to instantiate the blocks connected to the worker according to the session's data. There is no standard way to do this in GR: we could use the techniques described in section 2.2.3 or we can come up with a new solution. Here, we choose to use the callback mechanism again: the callback function is called from the worker with session data, so the top_block can (re)instantiate the GSM receiver blocks.

3.4 Signal compression

Complex data are typed gr_complex, which is a redefinition of std::complex in GR. Complex number are represented as two consecutive single-precision floating point numbers, thus having 64-bit. The USRP ADC has a precision of 14 bits, so theoretically we can store complex samples in 28 bits without any precision loss. However, the samples go through a processing chain and are even processed by the FPGA before coming to the UHD source. GR provides no facility to handle data types of different precision, and there is no surprise here. Most architecture have floating point unit (FPU) for single-precision floats and rearranging/convert data would introduce a massive performance penalty. However, when using an efficient channelizer, the system is now limited by the link capacity when using GbE: we are bounded at about 15.6 Msps which is roughly 57 channels.

What we can do to improve the system without upgrading to a 10GbE connectivity is compressing the signal. Indeed, we cannot convert back our samples to 24 bits since they already went to a processing chain: it would introduce precision lost and this is unwanted for our application. However, we can try to use generic compression algorithms. As for any compression algorithm, the worst case scenario would be that we have random data. In other words, it means the signal samples are evenly distributed in their floating-point representation. However, this situation is unlikely: we start from lesser-precision numbers and then the processing chain is limited. Therefore, we expect to achieve significant space saving using generic compression algorithms. Using sampled GSM downlink signal data captured from the USRP after channelization, we get the following compression ratio using popular compression algorithms:

Name	compression
Gzip	48.4 %
bzip2	36.6 %
Lzma	44.4 %
lz4	78,2 %

Table 18 – Compression ratios on digital GSM signals

As we can see, this is possible to achieve fairly good compression ratio on our signal data. There is also some big differences between the algorithms: they have different space-time tradeoffs. Let's evaluate the compression/decompression speed to see if one of these algorithms might suit our requirements:

Name	Compression speed	Decompression speed
Gzip	39.2 MB/s	87.7 MB/s
bzip2	9.68 MB/s	24.9 MB/s
Lzma	5.74 MB/s	29.3 MB/s
lz4	155.6 MB/s	526.7 MB/s

Table 19 – Measurements on compression speed

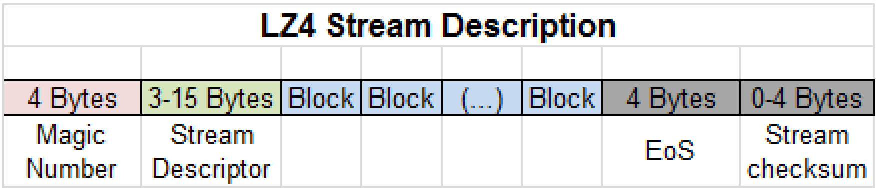
We performed these tests five times each on an Intel® Core™ i5-4200U CPU @ 1.60GHz using the following CLI commands:

- ❏ time gzip -1 gsm_1ch_959MHz.dump; time gunzip gsm_1ch_959MHz.dump.gz
- ❏ time bzip2 -1 gsm_1ch_959MHz.dump; time bunzip2 gsm_1ch_959MHz.dump.bz2
- ❏ time lzma -1 gsm_1ch_959MHz.dump; time unlzma gsm_1ch_959MHz.lzma
- ❏ time ./lz4c -c0 gsm_1ch_959MHz.dump; time ./lz4c -d gsm_1ch_959MHz.lz4

We used the lowest compression level setting (= 1) for testing, which trades compression ratio for better speed. Compression settings usually vary from 1 to 9, but the lz4 algorithms only have two compression levels.

We can observe that the compression ratio is directly proportional to the compression time, which is expected. Note that multithreading is not enabled for compression/decompression in our tests, which means that we can expect different results in real usage. It is clear that only the lz4 algorithm is fast enough for real-time data compression on > 1 Gbps data transfers. lz4 is a compression algorithm focused on compression and decompression speed. It is even sometimes advertised as "as fast as a *memcpy()*" and a library is freely available using the BSD license.

We implemented some optional (at compile-time) data compression inside our tcp_source and tcp_sinks. As lz4 was originally built for in-memory compression, there were no exchange format but a framing (or streaming) format was recently introduced. As the framing library is not considered stable yet, we tried to implement the framing logic but it proved to be very cumbersome. We finally decided to use the framing library <lz4f>. A lz4 frame is described in figure 20. As an exchange format, it can be used to communicate between various architectures.



LZ4 Stream Description							
4 Bytes	3-15 Bytes	Block	Block	(...)	Block	4 Bytes	0-4 Bytes
Magic Number	Stream Descriptor					EoS	Stream checksum

Figure 20 – lz4 stream description, lz4 documentation

4 High-performance polyphase filterbank channelizer

4.1 Signal channelization overview

4.1.1 Introduction

We previously stated that a multi-rate signal processing application typically consists of (1) an analog receiver - including analog filters, (2) a digital front end - placed right after the ADC and (3) a DSP application [7]. The digital front end contains the frequency shifting and filtering processing blocks. The pass filters, which split the signal into channels, constitute together a filterbank. Because this processing block comes directly after the ADC, it needs to operate at the same sampling rate, which can be very high. For this reason, this process is very computationally intensive. In industrial applications, the filtering process is usually done using dedicated hardware. In our GSM test system, the digital front end channelizes the signal to equi-spaced GSM channels and eventually down-samples individual channels to match the DSP rate (~270ksps in our case). When building distributed SDR applications, we require the digital front end to operate fast enough to avoid becoming a limiting factor of the overall system.

4.1.2 Pass filters

Pass filters (in the frequency-domain) are typically implemented using finite impulse response (FIR filter) or fast Fourier transform (FFT filter) [8]. We will not describe FFT filters in detail as we will not require them.

A FIR filter is an accumulator function known as discrete convolution. It is illustrated in figure 21.

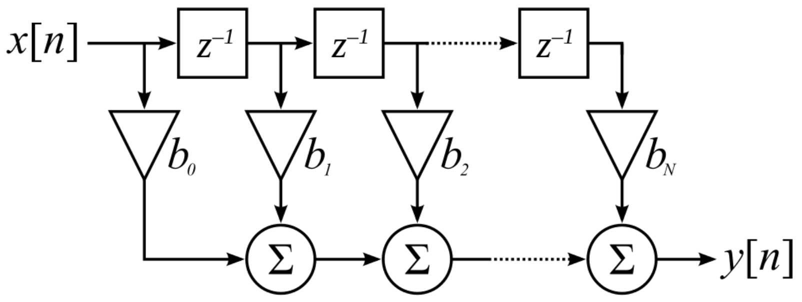


Figure 21 – FIR filter, BlanchardJ[17]

It is formalized as:

$$\mathrm{y[n]} = b_0\mathrm{x[n]} + b_1\mathrm{x[n-1]} + \ldots + b_N\mathrm{x[n} - \mathrm{N]} = \sum_{i=0}^{N} b_i \cdot x[n-i]$$

[17] http://commons.wikimedia.org/wiki/File:FIR_Filter.svg

where

- ❑ $x[n]$ is the input signal at discrete time n
- ❑ $y[n]$ is the output signal at discrete time n
- ❑ b_i are the weights (coefficients of the filter)
- ❑ N is the filter order (amount of past input samples to remember)

GR includes classical FIR and FFT filters, as well as frequency XLating versions. Freq. XLating filters performs frequency shifting, pass filtering and down-sampling (decimation) in one stop. They are also more computationally efficient than processing the signal using three dedicated GR blocks.

FIR filters are faster when having a low number of taps. For high number of taps and depending on the implementation, FFT filters are outperforming[18,19] FIR filters. Any of both can be used when we do not care about performance.

We can analyze the complexity of channelizing the signal using simple FIR filters. When using N filters to channelize our M channels signal, each filter has an input rate equal to the ADC sampling rate. For a 5 channels GSM signal sampled at 1.35 Msps (about 271 ksps per channel), each filter has an input rate of 1.35Msps. An N-taps FIR filter performs $2 + 4(N_{taps} - 1)$ floating-point operations [5] using complex samples. It means that we have to handle $rate_{sampling} \times M_{filters}(2 + 4(N_{taps} - 1))$ floating point operations per second (FLOPs). For our 5 channels input signal, this represents $13.5(2 \times N_{taps} - 1)$ MFLOPs. If we want to capture 50 GSM channels, we need an input rate of 13.5 Msps. Using the proposed formula, we see that the complexity grows exponentially: it represents $1350(2 \times N_{taps} - 1)$ MFLOPs. To make things worse, we have to use more taps when dealing with more channels due to the wider input bandwidth. Indeed, the FLOPS do not measure exactly the overall performance as there are other types of operations involved; mainly memory transfers and integer arithmetic (loop(s) indices, pointers arithmetic). However, floating-point operations are the most computationally intensive operations of FIR filters. For this reason, FLOPS are a good indicator of runtime complexity.

For this reason, the GSM system was initially not able to handle more than about 15 channels simultaneously using our testing setup with the XLating filter. In the next section, we will describe an efficient channelization technique for equi-spaced channels.

4.1.3 Polyphase filterbank channelizer

A polyphase filterbank (PFB) channelizer is an efficient technique for splitting a signal into e-spaced channels [8]. This technique is perfectly applicable for GSM signal where we have 125 equi-spaced channels in both downlink and uplink. It is composed of M FIR filters (the filterbank) and an M-point (bins) FFT.

[18] http://gnuradio.squarespace.com/blog/2014/2/27/to-use-or-not-to-use-fft-filters.html
[19] http://www.dsprelated.com/dspbooks/sasp/FFT_versus_Direct_Convolution.html

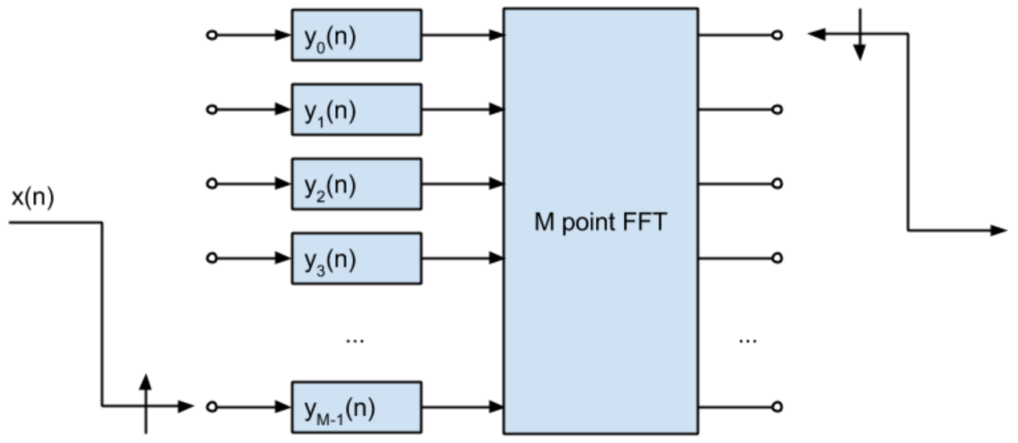


Figure 22 – PFB channelizer

We can now discuss the complexity of a PFB channelizer. As input signals are consecutively sent to next delay line (phase rotation), each FIR filter $\mathrm{y_m}$ has a delay line of $\frac{N_{taps}}{M_{filters}}$ items. The cumulated complexity of the FIR filters is hence:

$$rate_{sampling} \times M_{filters}\left(2 + 4\left(\frac{N_{taps}}{M_{filters}} - 1\right)\right) = rate_{sampling}(4N_{taps} - 2M_{filters}).$$

We should indeed add the operations performed by the M point FFT to measure the overall complexity. We did not analyze in detail the amount of FLOPs performed by an FFT but it is significantly lower than the filterbank [10], especially for a high number of taps.

If we calculate the amount of FLOPS of a polyphase filterbank using 5 and 50 GSM channels, we can demonstrate that the complexity is growing linearly. Using the 5 channels signal (1.35Msps), we get $1.35 \times (4N_{taps} - 10)$ MFLOPs. If we increase the band to 50 channels (13.5Msps), we get $13.5 \times (4N_{taps} - 100)$ MFLOPs. It is a huge increase in speed over the $13.5(2 \times N_{taps} - 1)$ MFLOPs for 5 channels and $1350 \times (2N_{taps} - 1)$ MFLOPs for 50 channels when using simple FIR filters for channelization (see previous section).

4.2 Implementation & optimization of a PFB channelizer

4.2.1 Overview

In Listing 2, we present simplified pseudo-code of a polyphase channelizer which requires that N_{taps} is a multiple of $M_{channels}$ so that $N_{taps} \times filter\ order_{FIR} = M_{channels}$.

```
Init()
    M <- m_channels
    N <- filter_order/m_channels
    Declare delay_lines[M, N]
    Declare m <- M-1 # fir filter index
    Declare d_idx <- 0 # delay index
Work()
    Do for i=0 to n_samples/M do:
        Do for m=M-2 to 0 do # mth filter, inverse rotation
        m <- Rotate(m) # M-1 if m=0 else m-1
            d_idx <- NextDelayIdx(m, d_idx) # each delay_line is a circular buffer
            delay_lines[m][d_idx] <- GetSample(i)
            fir <- 0
            Do for n=0 to N-1 do:
                tap <- delay_lines[m, n]
                weight = weights[m * N + n]
                fir += tap * weight
            ComputeFFT(delay_lines[m], dimensions=1, plan_size=M)
```

Listing 2 – PFB channelizer pseudocode

Note that the GR implementation of the PFB channelizer supports oversampling. We will not discuss the outcomes of this specific feature.

From the figure 22, it is pretty clear that this process can be easily parallelized, the only synchronization point being the input of the FFT.

Due to the vectorized nature of signal processing operations, CPUs are computationally not efficient compared to FPGAs, ASICs or even GPUs in this area. For this reason, architecture-specific implementation for intensive computation is important in SDR.

GNU Radio takes advantage of architecture-specific operations through the VOLK machine[20]. The Volk machine is a collection of GR libraries for arithmetic-intensives computations. Depending on the host architecture, an efficient native implementation can be selected at runtime. When no native implementation is available or if we are using an exotic architecture, Volk libraries provide a generic version of these functions that can be compiled in any architecture supported by the compiler. Most of the provided implementations are using x86 or ARM vector instructions (MMX, SSE_x, AVX, NEON). Vector instructions, also called single operation multiple data (SIMD) instructions, enable the same instruction to be performed on multiple data (vector input) to produce several results (vector output) in one step. These operations can improve the speed significatively when dealing with highly parallelized operations on vectors.

The FFT operations are computed through the popular fftw3[21] library. This open source library takes advantage of modern architecture by making use of specific SIMD operations. As we consider this library to be optimal for CPU usage when used correctly, we will not consider further FFT optimizations.

[20] https://gnuradio.org/redmine/projects/gnuradio/wiki/Volk
[21] www.fftw.org

4.2.2 Using SIMD instructions

As described previously, FIR filters are accumulators; meaning that they perform a lot of multiply-accumulate operations. For this reason, the built-in PFB channelizer relies heavily on the VOLK multiply-and-add routine and most of the CPU cycles are spent inside this function [4]. Our test setup CPU (Intel Haswell) is compatible with several x86 SIMD instructions sets, including MMX, SSE1-4 and AVX. Indeed, these instructions sets are not equivalent in terms of performance and features and older instructions set are available for compatibility reasons. As stated previously, the fastest instruction set is automatically selected by GR. The architecture can be also configured manually for specific cases or for performance comparison.

The fastest multiply-and-add routine available in the VOLK machine for our CPU is *volk_32fc_32f_dot_prod_32fc_a_avx*. Using VOLK standardized naming scheme, we can infer that it is a dot product of complex and float vectors, resulting in a complex vector. The name also specify the memory alignment ("a" for aligned) and the instruction set (AVX). In the next sections we will explain AVX instruction set and memory alignment.

4.2.2.1 Advanced Vector Extensions

Advanced Vector Extensions (AVX) are extensions to the x86 instruction set. They can be considered as an upgrade to the SSE instructions set, as most operations are similar with increased registers size and cache line width. They allow operations on 256 bit registers, enabling 8 single-precision floating point numbers to be stored side-by-side on the same register and processed simultaneously.

AVX uses relaxed memory alignments requirements compared to SSE instructions. It means that using unaligned data is allowed for most instructions[22], but this comes with a performance penalty[23].

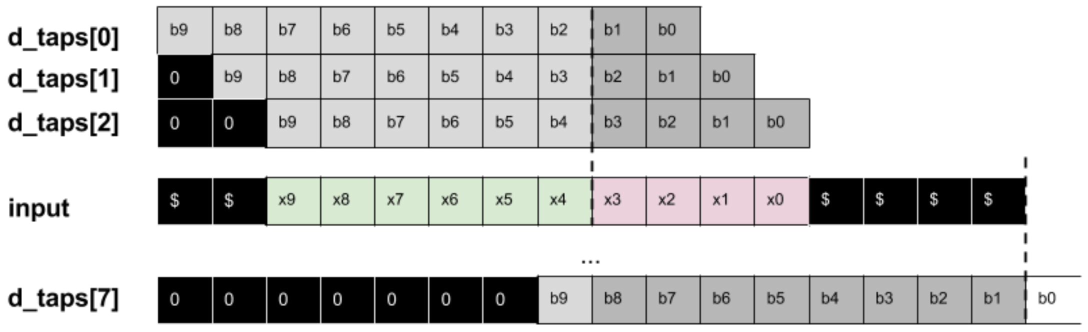
4.2.2.2 Memory alignment

CPUs accesses memory in chunks. Depending on the memory access granularity, a CPU may retrieve data in 2, 4, 8, 16, 32-bytes chunks or even more. Depending on the architecture of the specific instruction, it may or may not be allowed to load data from memory into register using addresses not multiple of the memory access granularity. Indeed, if allowed, loading unaligned data requires more operations as the register will be a compound of two different memory chunks merged. Outcomes of memory alignment are detailed in an IBM article[24].

When allocating memory using *malloc()*, compilers are in charge of the alignment, and we cannot assume that the returned address is aligned. However, it is possible to force alignment when desirable. Such a function allocates a slightly bigger amount of memory ($desired\ size + granularity - 1$) and moves the pointer to the next aligned address. GR provides such facility through *the volk_malloc()* function. However, the FIR filters from the PFB channelizer cannot use this function: GR buffers are not

[22] http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf - 14.9 MEMORY ALIGNMEMENT, p. 345
[23] https://software.intel.com/en-us/articles/practical-intel-avx-optimization-on-2nd-generation-intel-core-processors
[24] http://www.ibm.com/developerworks/library/pa-dalign/

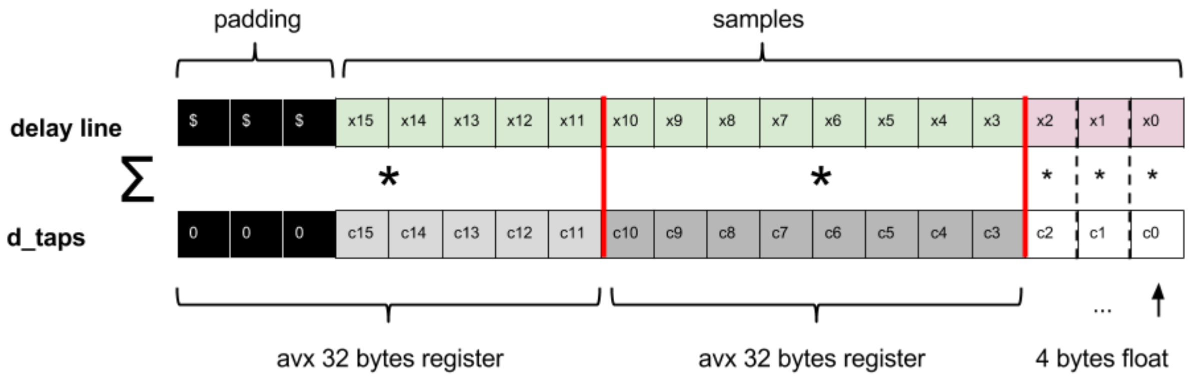
necessarily aligned. Instead, it generates a set of taps (taps are FIR weights in GR) for each possible architecture alignment. It is possible thanks to the properties of multiply-and-add: if we put zeros as weights, the effect will be similar as discarding input values. By measuring the alignment of the input samples, we can select the properly aligned taps and always use aligned data. In the illustrated example of figure 23, we have a FIR filter with a history of 10 samples. As a simplified example, let's say we have a 32-bit machine, and we do not use complex numbers. On our 32-bit machine, we have 8 possible alignments using 32-bytes memory access granularity. In this example, we have an input with a padding of two, meaning there is an offset of 8 bytes from the previous aligned pointer. With this knowledge, we can select the appropriate taps = d_taps[2] to perform the convolution operation on aligned data.



d_taps[0]
b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
d_taps[1]
0 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
d_taps[2]
0 0 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
input
$ $ x9 x8 x7 x6 x5 x4 x3 x2 x1 x0 $ $ $ $
...
d_taps[7]
0 0 0 0 0 0 0 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0

Figure 23 – GR FIR taps alignment

As discussed earlier, the *volk_32fc_32f_dot_prod_32fc_a_avx()* function uses aligned data. Using AVX instructions, we perform multiplications and additions on 256-bit vectors containing 8 floats each. If we do not have a history which is multiple of 8, the function process the remaining of the input using non-SIMD operations. It is illustrated in figure 24.



padding
samples
delay line
$ $ $ x15 x14 x13 x12 x11 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 x0
Σ
*
*
* * *
d_taps
0 0 0 c15 c14 c13 c12 c11 c10 c9 c8 c7 c6 c5 c4 c3 c2 c1 c0
...
avx 32 bytes register
avx 32 bytes register
4 bytes float

Figure 24 – GR FIR AVX dot product

4.2.2.3 Fused multiply-add (FMA) operations

Fused multiply-add (FMA) operations are floating point operations similar to x <- a + (b * c) that are performed in one step. Recent x86_64 CPUs provide such operations through the FMA3 or FMA4 (both are incompatible) instructions set. FMA4 was introduced by AMD in Bulldozer architecture while FMA3 is included in AVX2 (Intel Haswell and onwards). AMD CPUs starting from Piledriver also support FMA3 for compatibility reasons. FMA3 means three-operands fused multiply-add, translating into the use of three registers. The left operand x from x <- a + (b * c) should be the same register than either a, b or c. FMA3 operations and intrinsics are extensively documented in [11]. These intrinsics are compatible with the GNU Compiler Collection (GCC). Benchmark of pure multiply and accumulate operations on Intel Haswell CPU shows a double increase of FLOPS using FMA compared to AVX multiply then add[25].

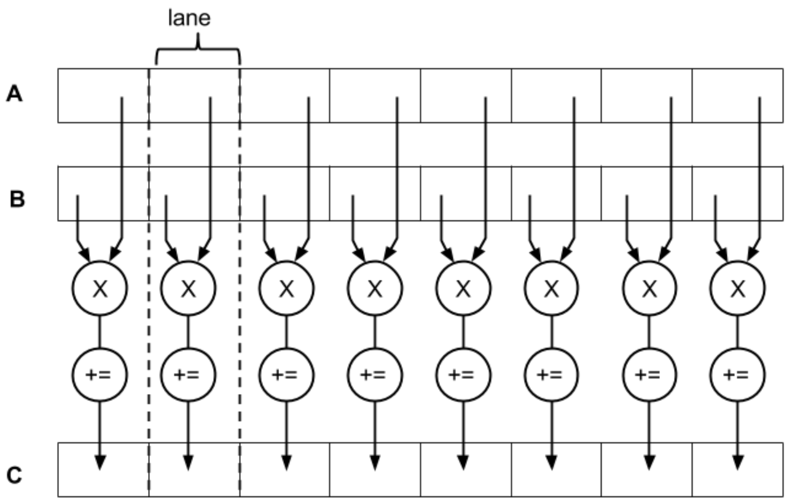


Figure 25 – FMA SIMD operation

In listing 3, we present the code for the dot product using FMA. For clarity and concision, the highlighted function is the product of floats vectors. The function we use in our filter bank is the product of complex and floats vectors, but it requires some further operations like complex input deinterleaving. The complete implementation for complex input is attached in appendix 1.

```
static inline void volk_32f_x2_dot_prod_32f_a_avx2(float* result, const  float* input,
const  float* taps, unsigned int num_points) {
  unsigned int number = 0;
  const unsigned int eighthPoints = num_points / 8;

  float dotProduct = 0;
  const float* aPtr = input;
  const float* bPtr = taps;

  __m256 aVal;
  __m256 bVal;

  __m256 cVal = _mm256_setzero_ps();

  for(;number < eighthPoints; number++){

    aVal = _mm256_load_ps(aPtr);
```

[25] http://stackoverflow.com/questions/21001388/fma3-in-gcc-how-to-enable

```
    bVal = _mm256_load_ps(bPtr);

    cVal = _mm256_fmadd_ps(aVal, bVal, cVal);

    aPtr += 8;
    bPtr += 8;
  }

  __VOLK_ATTR_ALIGNED(32) float dotProductVector[8];

  _mm256_store_ps(dotProductVector, cVal); // Store the results back into the dot product vector

  dotProduct = dotProductVector[0];
  dotProduct += dotProductVector[1];
  dotProduct += dotProductVector[2];
  dotProduct += dotProductVector[3];
  dotProduct += dotProductVector[4];
  dotProduct += dotProductVector[5];
  dotProduct += dotProductVector[6];
  dotProduct += dotProductVector[7];

  number = eighthPoints*8;
  for(;number < num_points; number++){
    dotProduct += ((*aPtr++) * (*bPtr++));
  }

  *result = dotProduct;
}
```

Listing 3 – VOLK AVX2 float dot product

We did also consider to take advantage of FMA operations for the FFT. While fftw3 is compatible with FMA operations[26], this feature is, by default, not enabled for AVX2 compatible x86 architectures. However, when compiling fftw3 using --enable-fma option and forcing the detection in the makefile, we did experience a negative impact on performance.

4.2.3 Further optimizations

It is possible to optimize the PFB channelizer further. Most of these modifications requires changes outside of the VOLK machine and make the implementation unsuitable for any other architecture, making the modifications unlikely to reach mainline GR.

4.2.3.1 Parallel filters

While we presented some very specific optimizations, we should keep in mind that until now the PFB implementation was purely sequential. We will describe in section 4.3 how we can do a highly parallel version running on a GPU, which can also be executed on the CPU. We can also make this implementation multithreaded, for example using OpenMP[27], we can take advantage of the "parallel for" pragma. It will,

[26] http://www.fftw.org/faq/section2.html#fma
[27] http://openmp.org/

however, require to remove or modify the oversampling feature due to its implementation and complexity.

4.2.3.2 Taps length always multiple of SIMD register

When using a filter order which is multiple of 8, the discrete convolution operation can be performed using only SIMD operations, removing the necessity to process the remaining inputs one float at a time. It can be achieved by adding zeros at the end of our aligned taps, but this requires code modifications. A better solution is to design our filter specifically for this length. Depending on our setup, this means we can compute more taps and have faster operations for free. Indeed, longer FIR filters mean higher delays, but increasing the length up to 7 samples is neglectable in our case.

4.2.3.3 Better pipelining

Our accumulator function is multiplying complex numbers by floating-point numbers. Indeed, there is nothing such as complex numbers for a CPU; complex numbers are just two consecutive floats as defined in *complex.h* of the C standard library (redefined as the *gr_complex* type in GR). Complex arithmetic rules define $(a + bi) \times c = (a + bi) \times (c + 0i) = (ac + bci)$, so we need a memory layout with every coefficient duplicated. As the GR FIR filters are not specifically for AVX operations, the coefficients are simply stored as consecutive floats. Unfortunately, feeding 256-bit registers with such a layout is a costly operation due to the available instructions. The best we can do is to load, unpack[28] and then permute[29], as shown in the code snippet below.

```
x0Val = _mm256_loadu_ps(bPtr); // t0|t1|t2|t3|t4|t5|t6|t7
x0loVal = _mm256_unpacklo_ps(x0Val, x0Val); // t0|t0|t1|t1|t4|t4|t5|t5
x0hiVal = _mm256_unpackhi_ps(x0Val, x0Val); // t2|t2|t3|t3|t6|t6|t7|t7

b0Val = _mm256_permute2f128_ps(x0loVal, x0hiVal, 0x20); // t0|t0|t1|t1|t2|t2|t3|t3
b1Val = _mm256_permute2f128_ps(x0loVal, x0hiVal, 0x31); // t4|t4|t5|t5|t6|t6|t7|t7
```

Listing 4 – VOLK AVX taps pipelining, GNU Radio source code

The behavior of unpack does not allow our coefficients to be extracted directly in the right order. Instead, the unpack operation uses 128-bit boundaries (figure 26). It is due to this operation directly inheriting from the legacy SSE unpack. The "trick" is to combine unpack with permute to reshuffle coefficients in the right order, but there is a certain overhead when doing this. What we can do to avoid this operation is to either:

- store directly duplicated coefficients
- shuffle the taps during creation

Storing duplicated coefficients requires doubling the memory accesses, which involves slow operations. Therefore, we expect this solution to be worse. However, by pre-shuffling the taps during the

[28] https://software.intel.com/en-us/node/514218
[29] https://software.intel.com/en-us/blogs/2015/01/13/programming-using-avx2-permutations

creation of the PFB, we can create a layout such that they are in the right order when unpacked, eliminating the need for a permutation.

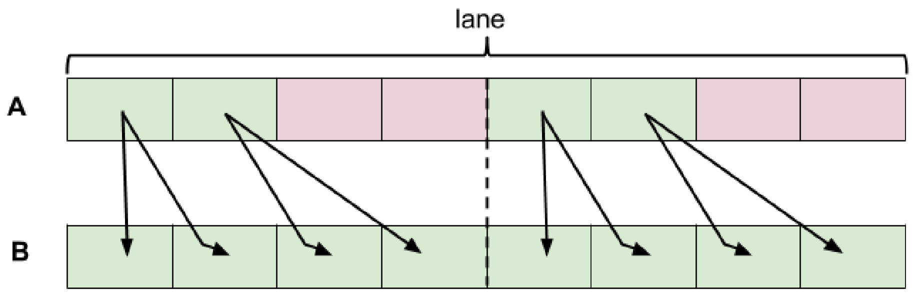


Figure 26 – AVX unpack operation

4.2.3.4 Loop unrolling

If we are in a situation where we always use the same filter in a static configuration, we can do an optimization technique called loop unrolling. As we always know how much multiply-add operations we will need to compute, we can statically do it multiple times, and thus eliminates instructions that control the loop. We can also use this technique to reduce loop control instructions. For example, we can loop over 16 points instead of 8 points points (AVX registers contains 8 floats, see appendix 1 for the loop). This is exactly what the VOLK AVX *dot_prod* function is doing. We will see in next section that it is a bad behavior for our PFB. Note that it is important to use a multiple of 2^N so the compiler can optimize the costly division operation to a bit shifting operation.

4.2.3.5 Pointer aliasing

As we do not delegate optimization to the compiler when using SIMD instructions manually, we do not need strict aliasing rules.

4.2.4 Experiments

In these experiments, we want to measure the speedup provided by FMA instructions compared to simple AVX instruction in the GR PFB channelizer. We also want to demonstrate that the loop unrolling optimization of the original *volk_32f_x2_dot_prod_32f_a_avx* function has a negative impact on performance for the PFB channelizer. All these tests are run on the testing setup presented in table 27.

Testing setup	
CPU	Intel® Core™ i7-4790 @ 3.60Ghz
Memory	32GB

Table 27 – AVX2 channelizer testing setup

To test the speedup provided by FMA, we use a simple benchmark illustrated in figure 28. The signal generator we use is fast enough not to interfere with the results and is widely used for GR benchmarks. It is preferred to file sources (using sample data as input) due to the negative impact of I/O operations on performance. The head block stops the input after n items were processed, so we can configure our benchmark depending on how long we want to execute it.

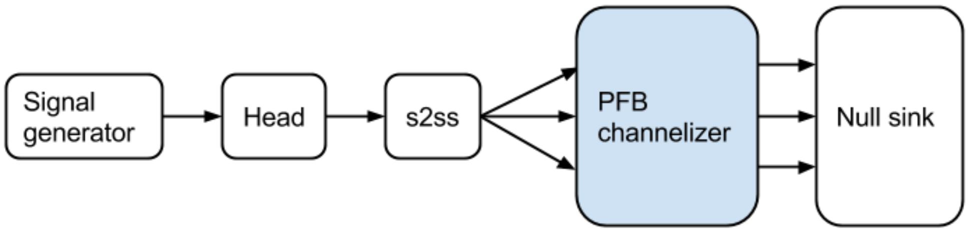


Figure 28 – PFB channelizer benchmark overview

First, we want to compare the performance of the PFB channelizer using our AVX2/FMA routine vs. the reference GR AVX implementation under normal conditions. We use here a pre-designed filter of 55 taps, and we want to test the performance for different numbers of channels. We generate a signal equivalent to 60s of data capture for an appropriate number of channels. The benchmark output is the total runtime, and the presented results are the arithmetic mean of 5 runs. See appendix 2 for complete Python benchmark.

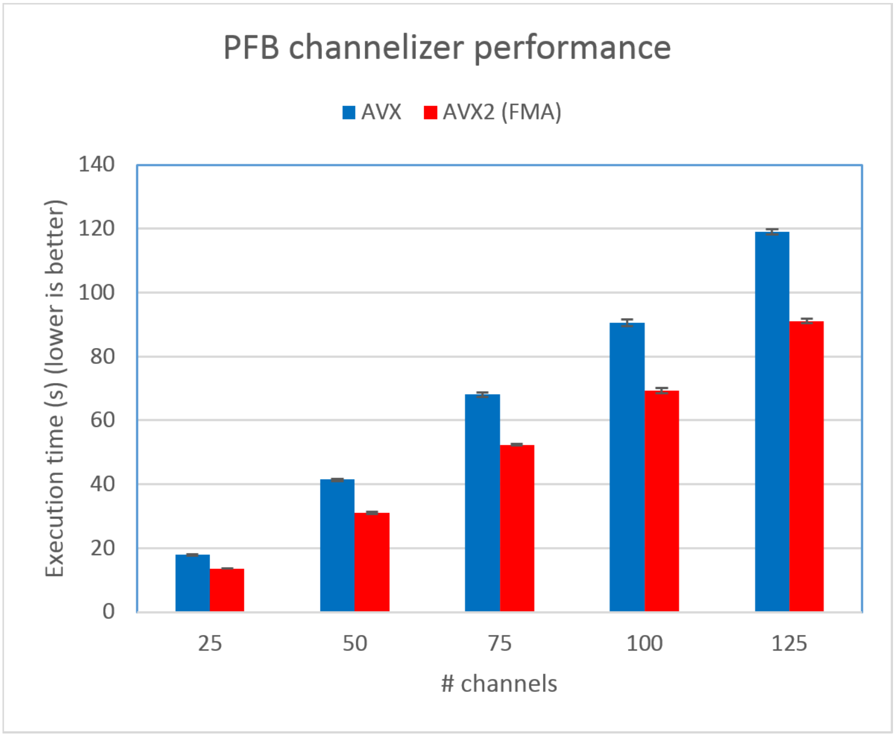


Figure 29 – PFB channelizer performance chart

If we look at the results from the chart 29, we notice quite significant improvements. The speedup ranges from 30% to 33%. It is, however still, not good enough for real-time GSM data analysis using the whole USRP bandwidth. The error bars delimit a 99% confidence level. As we can see, there is very few variation.

Now let's run the same benchmark (for 25 channels) but with varying taps per channel (filter) order. In the chart 30, we can clearly see the impact of loop unrolling. The situation with 55 taps is very advantageous for AVX2, so we may believe it mitigates the result from previous benchmarks. Howver, it was not arbitrarily chosen as it is the real filter design used by our GSM system. While long delay lines may benefit from it, PFB filters delay lines are short as the filter order is $\frac{N_{taps}}{M_{channels}}$. Filter order multiple of 16 does not suffer from it, but there is a huge improvement when$filter_order \bmod 16 \geq 8$. Due to AVX operations always processed in pair (2×8) in the reference implementation, extra samples are processed using standard non-SIMD operations. In our implementation, we benefit from an extra AVX operation. However, as *volk_32f_x2_dot_prod_32f_a_avx* is used by a wide range of blocks, short inputs may rarely happen, so the reference implementation makes sense.

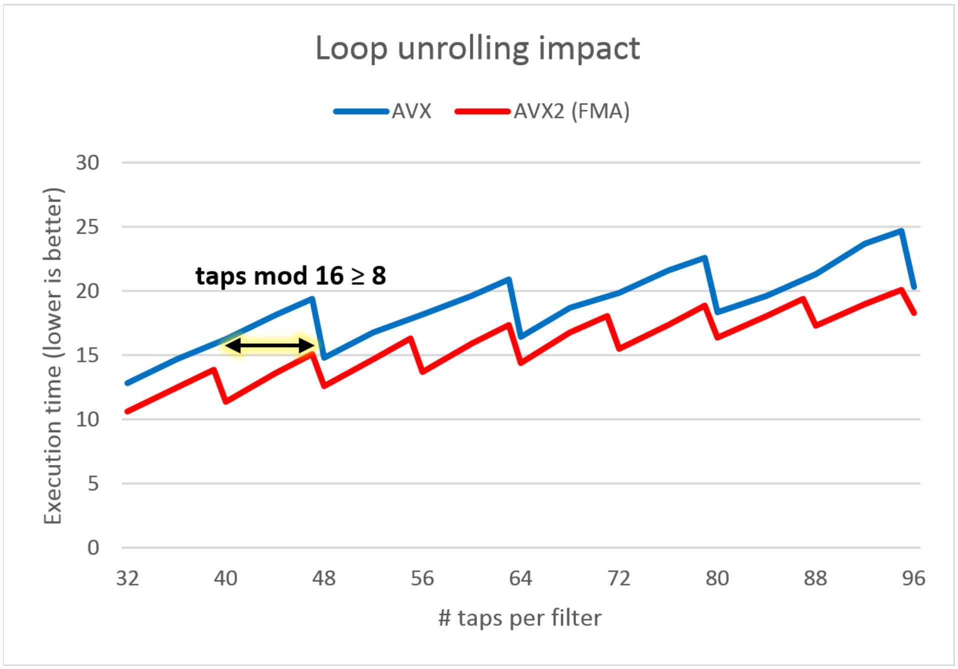


Figure 30 – Loop unrolling impact on PFB channelizer

4.3 GPU-accelerated implementation of a PFB channelizer

4.3.1 Overview

While we demonstrated that we could implement a fast PFB channelizer in a modern CPU, this element is still the bottleneck of our system. We are still not able to process the full band the USRP is capable to capture. We can, however, make this processing faster by taking advantage of massively-parallel architectures.

While there are numbers of massively-parallel architectures, including FPGAs, graphic processing units (GPUs) and coprocessors; we will focus on GPUs. GPUs are widely available, cheap and already available in our testing setup. Compared to other specialized architectures, they are also relatively easy to take advantage of due to the various efforts of several GPUs vendors on general-purpose computing on graphics processing units (GPGPU). Thanks to programmable shaders, GPUs are not only able to perform a fixed set of graphics-related operations, but also general-purpose processing as we would expect from a general-purpose processor like a CPU.

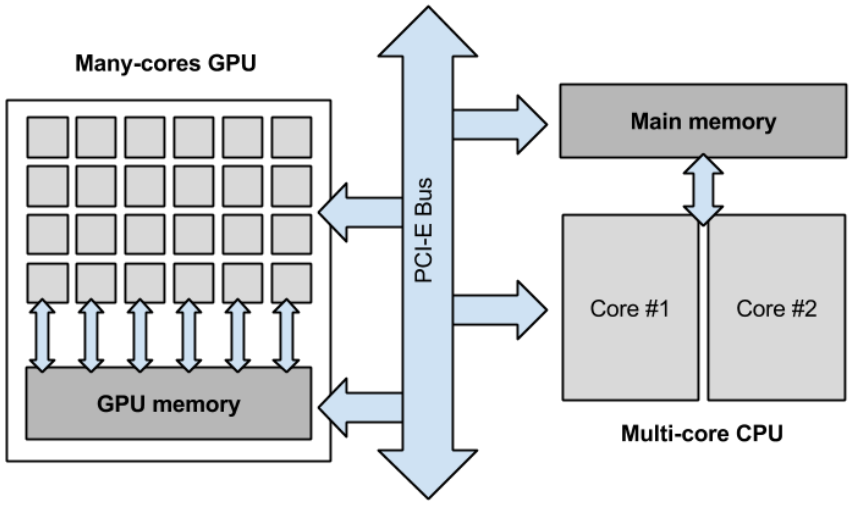


Figure 31 – Memory transfers with discrete GPU

They are several technologies available to do GPGPU. CUDA is a programming model created by NVIDIA and implemented in their GPUs. Since CUDA was released in 2007, GPGPU gained much popularity since it greatly simplified GPGPU programming. OpenCL is an open framework for heterogeneous computing launched in 2009 by Khronos and co-designed by Apple. OpenCL is vendor-neutral and can be used with CPUs, GPUs, FPGAs or even DSPs. In practice, CUDA and OpenCL programming are pretty similar in and there are even tools to translate CUDA code to OpenCL as most of the keywords have equivalents. Microsoft DirectCompute is a Windows-only API for GPGPU available on DirectX 10/11 cards. There are also various efforts to standardize heterogeneous computing like Heterogeneous System Architecture (HSA), but these are out of our focus.

Since we are running on Linux, we can choose to use CUDA or OpenCL. While CUDA is tied to NVIDIA cards, OpenCL platforms are available on all major platforms; including AMD, NVIDIA, and Intel GPUs, as well as Intel and AMD x86 AVX-enabled CPUs. It is also more open by design and, therefore, best suited to the philosophy of GR. For this reason, we will implement a PFB channelizer using OpenCL.

There are some ongoing discussions about adding coprocessor support to GR[30]. Unfortunately, this feature is not available yet. There is also a fork of GR called GRAS[31] featuring a complete rewrite of the

[30] https://gnuradio.org/redmine/projects/gnuradio/wiki/Call20140812

[31] https://github.com/guruofquality/gras/wiki

scheduler and includes add-on support for OpenCL[32]. Unfortunately, the project is not maintained anymore. Other GR OpenCL-based projects include gr-theano[33], a project enabling OpenCL in Python blocks and gr-fosphor[34], a GPU-accelerated spectrum visualization tool.

4.3.2 The OpenCL model

OpenCL has been designed to fit heterogeneous architectures. For this reason, it provides an execution model that must fit various hardware implementations, including GPUs. OpenCL terminology includes:

- ❑ **Platform**

The platform is a vendor implementation of the OpenCL API and language. It includes a library, a kernel compiler and drivers. A platform is available for a set of devices, usually from the same vendor.

- ❑ **Kernel**

Parallel applications running on a device are defined inside a kernel. Kernels are written using a C99-based language and compiled for a particular platform, usually at runtime.

- ❑ **Device**

The device is where the OpenCL software is executed on (a GPU in our case). It is opposed to the host, which use the OpenCL APIs to communicate with the device. The device architecture is described in figure 32. Each kernel instance is executed on a work-item. It can be compared to a CPU thread. Work-items are grouped in a work-group, forming a compute unit sharing memory. Work-items in the same compute unit have premium relationship with other work items over synchronization and memory sharing.

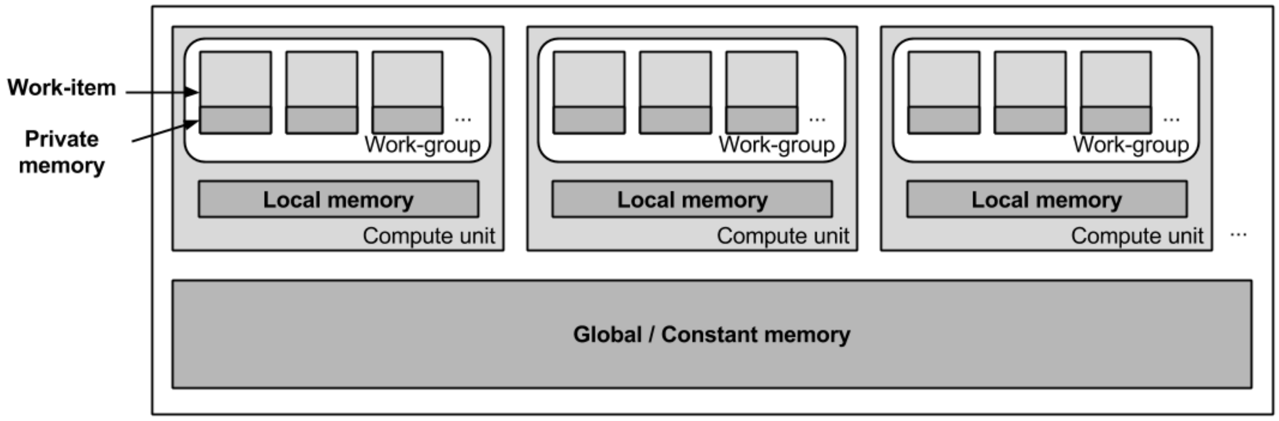


Figure 32 – OpenCL model

[32] https://github.com/guruofquality/grex/wiki/Opencl
[33] https://github.com/osh/gr-theano/tree/master/Python
[34] http://git.osmocom.org/gr-fosphor/

4.3.2.1 Memory spaces & access

OpenCL provides access to several memory spaces with different locality, size and transfer speed/latency properties:

- ❑ Global memory: globally shared memory
- ❑ Constant memory: read-only shared memory
- ❑ Local memory: work-group shared memory
- ❑ Private memory: work-item private memory

Private memory is very fast, but its storage space is very limited and is only accessible by the current work-item. Local memory is slower than private memory, but it is shared with other work-items of the same work-group. Global memory provides a lot of storage space but is the slowest memory of the device. Shared memory should be explicitly synchronized if necessary, as OpenCL uses a relaxed memory consistency model. Constant memory can be slightly faster than global memory depending on the platform.

4.3.3 Memory transfers

4.3.3.1 Transfer techniques

In our test setup, we are using a discrete graphic cards, meaning that the CPU and GPU do not share the same memory. The GPU is a high-throughput, high-latency OpenCL device. Quoting clFFT documentation's[35] networking analogy, "it is similar to having a massively high-bandwidth pipe with very high ping response times". There are several techniques to transfer memory between the CPU and GPU, with various transfer speeds [12]. Memory is always transferred from/to the device global/constant memory. When using physically shared memory between the GPU and CPU using fused architectures (Intel IGP, AMD APU...), we are not required to transfer memory. This technique is called zero-copy. For discrete GPU, the memory could be copied using the CPU or using GPU direct memory access (DMA). To use the DMA feature, the memory should not be paged out by the operating system during the transfer. Memory regions that should never be paged out are called "pinned" memory. Depending on the situation and the amount of data to transfer, the data could be transferred using the CPU, copied to an already allocated pinned region then transferred using DMA, or the data may be pinned directly then transferred using DMA and then unpinned. How the memory is actually copied is platform-dependent and totally transparent to the programmer. For AMD platforms, a list of transfer techniques depending on the situation is available on developer documentation[36]. Indeed, how the memory is transferred has a high impact on transfer speeds. The following table from AMD shows reference transfer speeds for their platform:

[35] http://clmathlibraries.github.io/clFFT/

[36] http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/#50401315_pgfId-503627

Memory Bandwidth in GB/s (R = read, W = write) in GB/s [12]

	CPU R	GPU W	GPU Shader R	GPU Shader W	GPU DMA R	GPU DMA W
Host Memory	10 – 20	10 – 20	9 – 10	2.5	11 – 12	11 – 12
GPU Memory	.01	9 – 10	230	120 -150	n/a	n/a

Figure 33 –OpenCL reference transfer speed, AMD

We want to be able to process data in real-time. Therefore, we have strict requirements for transfer speeds. As previously described, GR blocks are connected through buffers which are managed by the GR scheduler. These buffers are contiguous memory stored in the main memory, which are in turn consumed/feeded by GR blocks in the *work()* function. Since GR has no built-in support for co-processors, buffers could only be transferred to/from device inside the *work()* function. Every *time work() is* called, we should (1) transfer from the main memory to the GPU, (2) performs the computation on the device and (3) transfer the result from the device to the main memory. This situation is not optimal as the (1) and (3) memory transfers have high latency but are used to transfers few data. An optimal solution would be to do simultaneously (3), (2) and (1) for consecutive inputs [6], but this is not possible in GR. GREX tackles this problem by adding buffer copying feature inside the scheduler[37]. A possible workaround for vanilla GR would be to implement (1) and (3) in different blocks (respectively sink and source) with synchronization mechanisms but we did not explore this idea.

4.3.3.2 Memory layout

Before discussing the implementation, we should define how we will transfer our data to/from the device and in which layout configuration. We choose to delegate deinterleaving (the rotator in the front of the PFB channelizer) to a stream_to_streams[38] block placed before our PFB channelizer, so we get the inputs of the FIR filters on different ports. As different ports use different buffers, we are required to do multiple transfers to the device using this solution. As it is recommended for performance to send as few API calls as possible to the device, we can batch the transfers using non-blocking successive calls to *clEnqueueWriteBuffer()*[39].

On the device, our input data is copied to the global memory. This memory is allocated once using an adequate maximum buffer size. Here we allocate memory for $M_{filters} \times (filter\ order + buffer\ size)$ complex numbers. Our taps are copied to the constant memory which is valid through the entire lifecycle of our PFB channelizer. We store the output of the FIR filters such that we can perform in-place FFT in batch. Input and output memory layout of the device are detailed in figure 34.

[37] http://www.joshknows.com/blog/51/FlexibleDataFlowBufferManagement
[38] http://gnuradio.org/doc/doxygen/classgr_1_1blocks_1_1stream__to__streams.html
[39] https://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/clEnqueueWriteBuffer.html

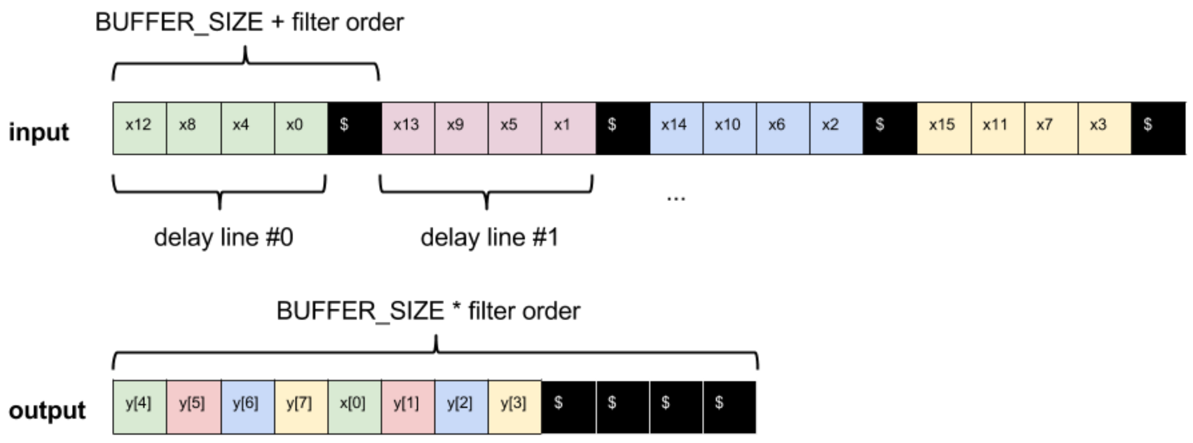


Figure 34 – PFB OpenCL memory layout

4.3.4 Implementation

What we want is to compute our accumulator functions on the GPU. These are simple multiply and add, so our GPU kernel is very simple. The implementation complexity of our channelizer lies in the host application, our pfb_channelizer GR block. The host handles OpenCL context creation, buffer management, and memory transfers as well as kernel execution calls. The FIR can run in parallel with each input samples in an independent work item. As we always process a multiple of $M_{channels}$, we run our FIR kernel $M_{channels} \times N_{output\ items}$ times, which we can specify in two dimensions in *clEnqueueNDRangeKernel*()[40].

The kernel source code is specified in listing 5. Here we get our filter index from the $M_{channels}$ first dimension and the delay index from the $N_{output\ items}$ second dimension. Input complex samples are A, and weights (taps) B. Note that each weight was duplicated in advance, such that they are stored in a w_0, w_0, w_1, w_1, ... w_{N-1}, w_{N-1} fashion. Similarly to the AVX FMA instruction, we can take advantage of the *fma()* or the *mad()* operations, a faster alternative without strict precisions guarantees[41]. Whether or not the hardware takes advantage of these instructions is platform/device dependent.

```
__kernel void fir_kernel(__global float2 *A, __constant float *B, __global float2 *C, int delay_len)
{
    //Get the index of the work-item \n"
    int filter = get_global_id(0);
    int delay = get_global_id(1);

    float2 dotProduct = (float2)( 0, 0 );

    __global float2 *delay0 = A + (8192+delay_len) * filter + delay;
    __constant float *w = B + delay_len * filter;

    for(int i=0; i<delay_len; i++){
```

[40] https://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/clEnqueueNDRangeKernel.html
[41] https://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/mad.html

```
        dotProduct = /*fma*/mad(delay0[i], w[i], dotProduct);
    }

    int nfilts = get_global_size (0);
    C[nfilts*delay + filter] = dotProduct;
}
```

Listing 5 – PFB FIR kernel source code

4.3.5 FFT

Once we get the output of the FIR filters, we have to compute the M-point FFT. We can choose to compute the FFT on the CPU or the GPU. As a highly parallelizable task, FFT can run significantly faster on the GPU. In addition, as the FIR output is already located in the GPU memory, so we have no performance drawback related to memory transfers.

There are several OpenCL FFT libraries available like AMD clFFT[42] and Apple FFT library[43]. To our knowledge, clFFT is the most feature-complete library. Besides being specifically optimized for AMD platforms, it is supposed to be compatible with other vendor's platforms as well. Moreover, it is open source and free-to-use. For these reasons, we choose to use clFFT.

clFFT usage is straightforward. Here we take advantage of the functions *bakePlan()* to apply device optimizations and *setPlanBatchSize()* to handle several plans concurrently, thus minimizing OpenCL API calls.

4.3.6 Verification

Our OpenCL implementation is almost compatible with the built-in PFB channelizer. It does not have the oversampling feature and requires $M_{channels}$ to be any mix of powers of 2, 3 and 5 due to a ClFFT limitation[44]. It otherwise uses a 1:1 interface. We therefore were able to compare the output of our implementation to the reference implementation under various scenarios. We can affirm that our implementation is 100% correct.

4.3.7 Experiments

Now we want to compare the performance of our newly implemented OpenCL channelizer to the reference AVX and modified FMA implementation. We can reuse the benchmark and the 55-taps filter proposed in previous experiments. We are reusing the test setup with an AMD R9 290 (Tahiti) GPU. The testing setup is described in table 35.

[42] https://github.com/clMathLibraries/clFFT
[43] https://developer.apple.com/library/mac/samplecode/OpenCL_FFT
[44] http://clmathlibraries.github.io/clFFT/

Testing setup	
CPU	Intel® Core™ i7-4790 @ 3.60Ghz
Memory	32GB
GPU	AMD R9 290
Video memory	4GB
Compute units	40
Stream processors	2560

Table 35 – OpenCL channelizer testing setup

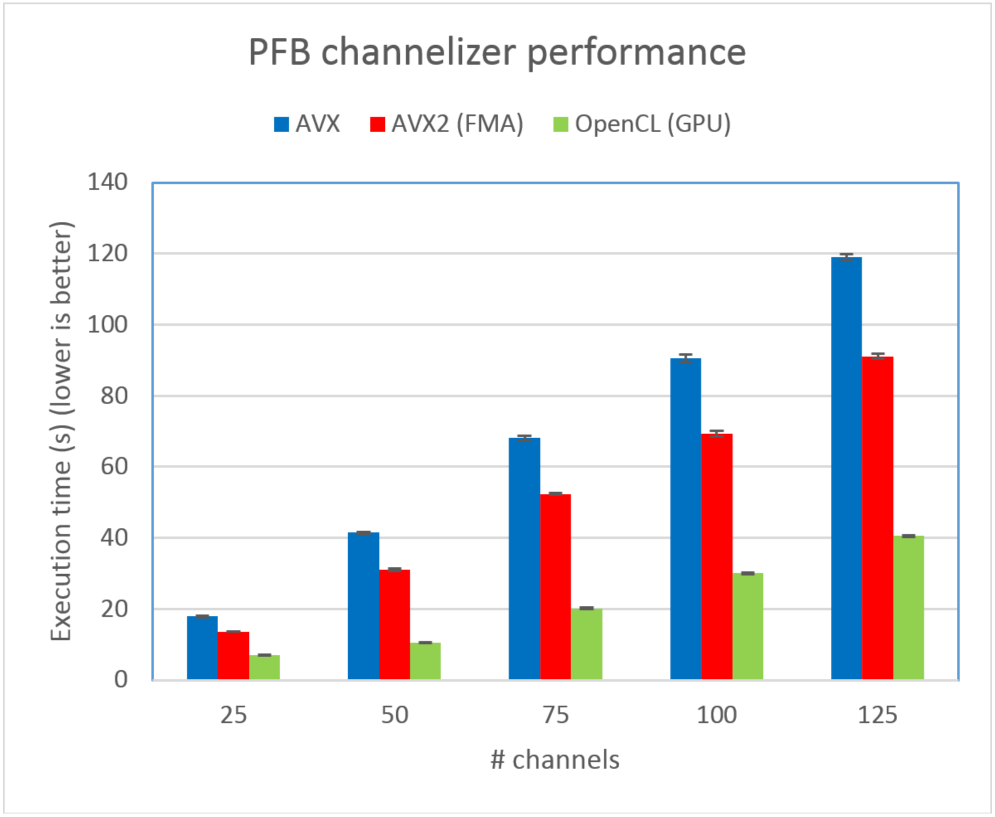


Figure 36 – PFB channelizer GPU performance chart

The results presented in figure 36 are impressive: compared to the reference channelizer, there is an average speedup of 3.2x. That means we are able to channelize more than 125 GSM channels, more than what the USRP is capable to capture.

5 Evaluation

5.1 Network module throughput

To validate the performance of our network modules, we require some high-capacity links to see if we are using the whole capacity or is our implementation limited by internal resources contention. To do this, we measure the throughput between our TCP blocks. As we are using one TCP connection (session) per channel without being dependent on the amount of workers, we want to evaluate what is the impact of multiple session on the overall throughput. As we are just interested in testing the network performance, the client simply sends some unuseful data, which are then discarded by the workers. There is no throttle block: the client sends as much data as he can to the workers. We test both the synchronous and the asynchronous implementations of the tcp_sink blocks.

5.1.1 Testing setup

For this experiment, we create 2 Amazon EC2 c4.4x large instances. We used these as they have an advertised dedicated throughput of 2,000 Mbps[45], more than the bandwidth required by the host to transmit 25 Msps of uncompressed 64-bit complex samples. We create a small set of workers in one instance and one client on the other instance. The load-balancer is used to simplify the setup, but this has no impact on the results.

Testing setup	
Configuration	Amazon EC2 c4.4x large
vCPU	16
MTU	9001 bytes (Jumbo frame)
Measured throughput[46]	2.18 Gbits/sec

Table 37 – TCP blocks throughput testing setup

To see how much traffic is generated, we use the network traffic tool vnstat. It has the advantage of requiring no code modifications while providing useful unaltered data. For each test, we measured the average throughput over one minute.

[45] http://aws.amazon.com/ec2/instance-types/

[46] we measured the throughput between two EC2 c4.4x large using the iperf tool

5.1.2 Results

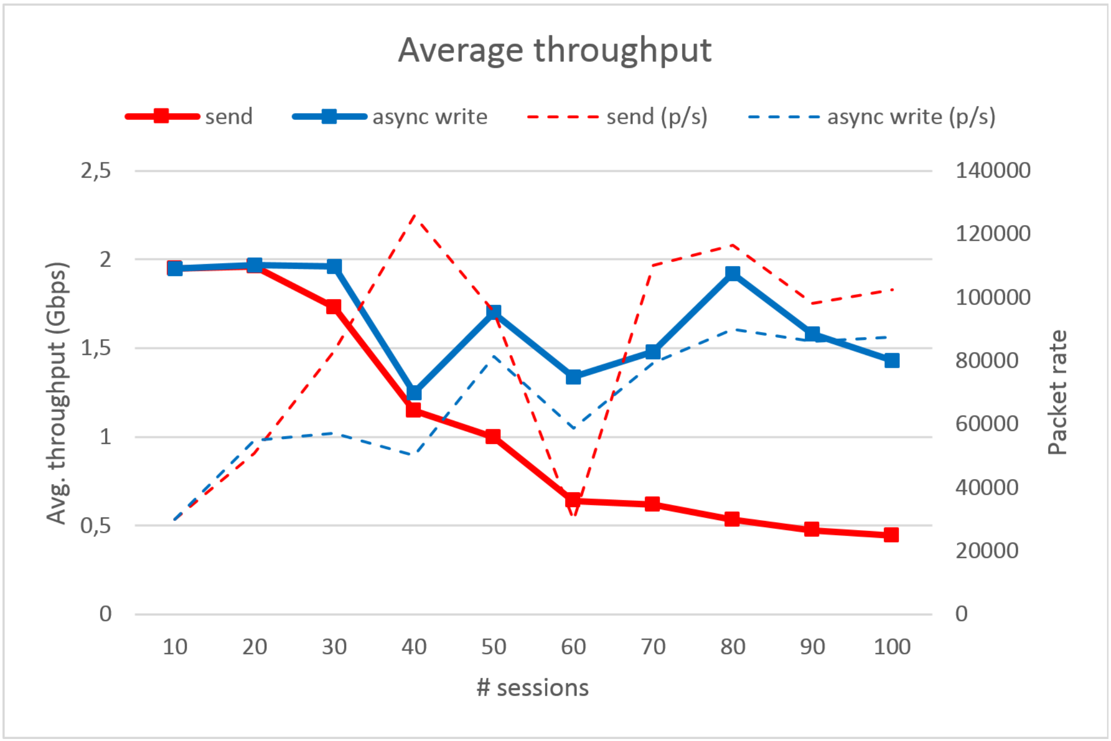


Figure 38 – TCP blocks throughput chart

We can observe in chart 38 very different behaviors between the synchronous and asynchronous implementation of the tcp_sink block. The performance of the synchronous version quickly declines with the number of sessions. It can be explained by the underlying boost ASIO implementation. According to a performance analysis[47, 48] of the ASIO implementation, the Linux version uses a synchronized wrapper around the kernel epoll system call. As each block runs on its own thread in GR, it creates a lot of contention. In opposite, asynchronous implementation shares one or more io_service thread (one in this test) responsible to handle socket i/o. IT completely solves the contention issue. We implemented the asynchronous as a result of this analysis and completely dropped the initial version.

If we analyze the evolution of the throughput of the asynchronous implementation, we can see that it is pretty inconsistent. Our hypothesis is that it is a consequence to how the GR scheduler works: the blocks should process an amount of data according to the flowgraph status. Under certain circumstances, it means that the amount of data processed by the tcp_sink does not behave well with the MTU. As a block may have to wait several milliseconds to be called by the scheduler again, it means that small or almost empty packets have to be sent as the socket is continuously filled with multiple sessions. It can be solved by going through an additional buffer. It is at the expense of additional memory usage and delay.

[47] http://cmeerw.org/blog/748.html#748
[48] http://cmeerw.org/blog/751.html#751

In addition, as the I/O operations are managed by one thread, we might benefit from a higher-frequency CPU.

As we require a goodput of $M_{channels} \times rate_{GSM} \times 64$, we can safely assume that our system support the transport of 90 channels with a using ≥ 2000 Mbps connectivity.

5.2 Integration tests

We want to validate our distributed architecture with the passive GSM receiver system. We will build a synthetic test using a uniform set of workers. While we tested the system using the USRP and heterogeneous workers, the performance is very difficult to evaluate due to the variety of configurations. The evaluation should highlight that the processed channels are scaling linearly until we reach a previously identified bottleneck.

5.2.1 Testing setup

We created a set of 8 Amazon EC2 m3 large instance workers which are responsible to capture GSM messages out of the channelized signal (see table 39). We use these as they have a dedicated throughput of about 500Mbps and moderate performance.

Testing setup	
Configuration	Amazon EC2 m3 large
vCPU	2
MTU	9001 bytes (Jumbo frame)

Table 39 – GSM receiver workers testing setup

For the host (client), we are reusing the Amazon EC2 c4.4x large instance from previous experiments. It has AVX2 capability so we can use the AVX2 PFB channelizer. Amazon provides instances for GPU compute applications but they do not have enough dedicated throughput for our application. Our c4.4x large instance is able to process up 76 channels using the AVX2 channelizer.

As we cannot use a real SDR receiver with this setup, we are using a signal generator. The signal obviously contains no GSM data, but this has very few impact on performance of the receiver.

5.2.2 Results

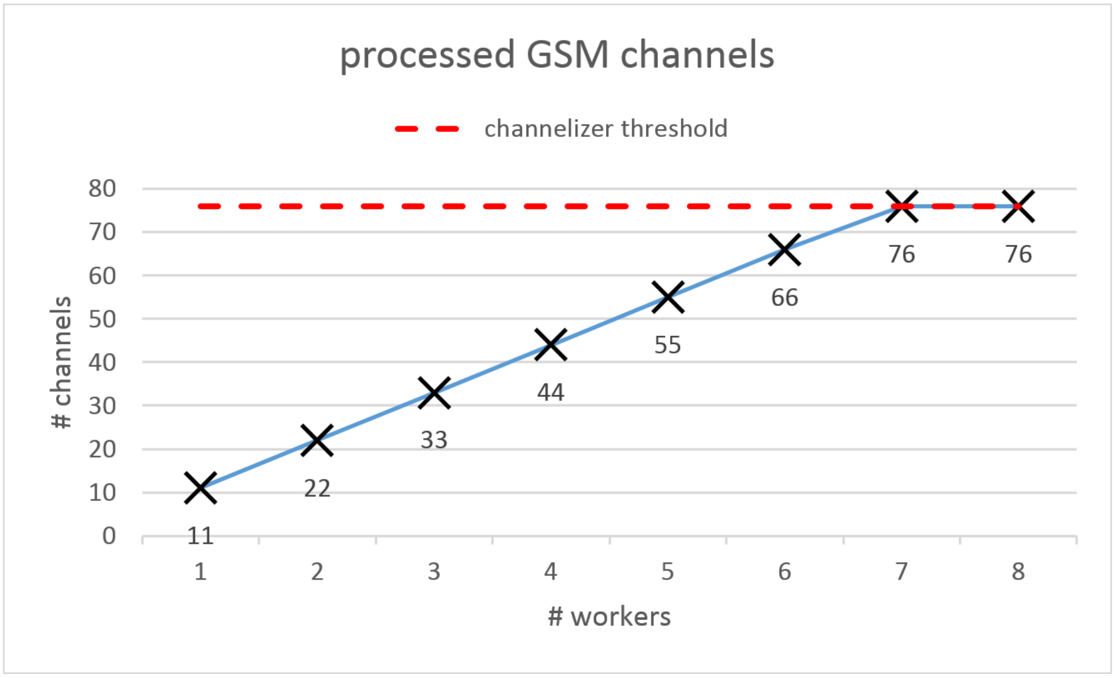


Figure 40 – TCP blocks throughput chart

In chart 40 we present the results from our evaluation scenario. The m3 large instances were able to analyze the digital signal from up to 11 channels. We can see that the processed channels are growing linearly when we add workers until we reach 76 channels. Note that there is no variation: the worker process a fixed amount of channels according to its performance.

The result is on par with our expectations. 76 channels is the maximum number of channels we can handle using a c4.4x large instance for the channelizer. We did not reach the throughput limit we previously evaluated.

6 Future work

6.1 Worker overload control

We initially planned to use contention algorithms to adjust the workload of the worker dynamically according to the available resources. While this feature would be a nice addition as it would add flexibility to the overall system and greatly simplify workers management, we refocused our efforts on removing the system bottlenecks as it was judged more important.

However, we can describe how we would implement it as it would require no architectural changes thanks to the token mechanism. In section 3.2, we described how workers can introduce tokens one by one to the load balancer. What we need is our worker to emit tokens according to the evaluated resources. An ideal worker would accept an amount of task that is (a) under a certain overload threshold and (b) above a reasonable occupancy level. If we are above (a), then our input buffer will overflow as we are not able to process the data as fast as it arrives. Similarly, if we are under (b), we are wasting some available resources. Consequently, we want that $b < \text{workload} < a$ and ideally $b = \text{workload} = a$. We do not want to end up in a situation where the worker is quickly switching between starvation and overflow: when a connection is dropped, the host needs to go through the token acquisition mechanism again. During this recovery period, the data is not kept in a buffer but simply discarded. For this reason, we better be safe and have reasonably low overload threshold.

If this feature is added at some point, we expect that it will be straightforward to make a working prototype. We also expect that it will be difficult to test and fine-tune it, so it runs optimally independently of the hardware it is executed on. They are several values a worker can use to take decisions, like CPU usage, (input) buffer level and recent buffer variations. Combining them in a meaningful way, so it is always optimal may be not achievable, but we think it is possible to obtain good results on a fixed set of hardware.

6.2 Worker status monitor

Another neat feature for workers' management would be a centralized worker status monitor. While there is few scientific value in such a tool, it would help much to manage a set of workers and to analyze their behavior. Using ZMQ, we can use heartbeating[49] mechanism to poll the workers regularly and display the result on a nice GUI interface. We can, for example, display the workers with their CPU load, buffers levels and any other information of interest.

[49] http://zguide.zeromq.org/page:all#Heartbeating

7 Conclusion

In this chapter, we present our conclusions and possible application opportunities regarding the work we did.

GR is the framework of choice for SDR projects. We demonstrated that it is possible to create high-performance, reusable components for creating a distributed system. However, many aspects of them can still be enhanced regarding the performance and the feature set.

While some basic building blocks to create a distributed flowgraph are already available (the network blocks previously mentioned as well as the experimental gr-zeromq module), they are barely suited to create dynamic and reliable high-throughput distributed applications. We introduce a solution including a load-balancer to create distributed GR applications. While we demonstrated the usage on a GSM system, this technique is applicable for any distributed flow graph that does not require cross-synchronization.

The polyphase filterbank channelizer is not only useful for our distributed system but is a critical component in many SDR systems using multiple channels. Our AVX2 optimizations provide an appreciable performance boost using a recent CPU, while the GPU implementation open up opportunities for applications were expensive specialized hardware was previously required. While many papers discuss about GPU PFB channelizer, there is few freely available implementations and, at the time of writing and to our knowledge, none of them is available for GR.

We successfully created a distributed wide-band receiver, giving the opportunity to resources-hungry SDR applications to have access to much more processing power. As we see other potential applications for this tools, we want to provide these tools to the open-source community in the form of GR modules.

8 Bibliography

[1] Alyafawi, I., Dimitrova, D. C., & Braun, T. (2014, June). Real-time passive capturing of the gsm radio. *In Communications (ICC), 2014 IEEE International Conference on* (pp. 4401-4406). IEEE.

[2] Marojevic, V., Reves, X., & Gelonch, A. (2005, September). Computing resource management for SDR platforms. In *Personal, Indoor and Mobile Radio Communications, 2005. PIMRC 2005. IEEE 16th International Symposium on* (Vol. 1, pp. 685-689). IEEE.

[3] Miranda, H., Carneiro, G., Pinto, P., & Silva, S. (2003, September). Implementation of Software Radio systems using distributed architectures. In *Sixth Baiona Workshop on Signal Processing in Communications*.

[4] Rondeau, T. W., Shelburne, V. T., & O'Shea, T. J. Designing Anaylsis and Synthesis Filterbanks in GNU Radio.

[5] van der Veldt, K., van Nieuwpoort, R., Varbanescu, A. L., & Jesshope, C. (2012, June). A polyphase filter for GPUs and multi-core processors. In *Proceedings of the 2012 workshop on High-Performance Computing for Astronomy Date* (pp. 33-40). ACM.

[6] Adhinarayanan, V., & Feng, W. C. (2013, December). Wideband channelization for software-defined radio via mobile graphics processors. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on* (pp. 86-93). IEEE.

[7] Pucker, L. (2003, November). Channelization techniques for software defined radio. In *Proceedings of SDR Forum Conference* (pp. 1-6).

[8] Harris, F. J. (2004). *Multirate signal processing for communication systems*. Prentice Hall PTR.

[9] *ZeroMQ zguide.* iMatix Corporation.

[10] del Mundo, C., Adhinarayanan, V., & Feng, W. C. (2013, June). Accelerating fast Fourier transform for wideband channelization. In *Communications (ICC), 2013 IEEE International Conference on* (pp. 4776-4780). IEEE.

[11] *Intel C++ Intrinsics reference.* Intel.

[12] OpenCL Optimization Guide. AMD. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/

9 Appendixes

VOLK AVX2 complex dot product

```c
static inline void volk_32fc_32f_dot_prod_32fc_a_avx2(lv_32fc_t* result, const lv_32fc_t* input, const float* taps, unsigned int num_points) {
  unsigned int number = 0;
  const unsigned int eighthPoints = num_points / 8;

  float res[2];
  float *realpt = &res[0], *imagpt = &res[1];
  const float* aPtr = (float*)input;
  const float* bPtr = taps;

  __m256 a0Val, a1Val;
  __m256 b0Val, b1Val;
  __m256 xVal, xloVal, xhiVal;

  __m256 cVal = _mm256_setzero_ps();

  for(;number < eighthPoints; number++){

    a0Val = _mm256_load_ps(aPtr);
    a1Val = _mm256_load_ps(aPtr+8);

    xVal = _mm256_load_ps(bPtr); // t0|t1|t2|t3|t4|t5|t6|t7
    xloVal = _mm256_unpacklo_ps(xVal, xVal); // t0|t0|t1|t1|t4|t4|t5|t5
    xhiVal = _mm256_unpackhi_ps(xVal, xVal); // t2|t2|t3|t3|t6|t6|t7|t7

    b0Val = _mm256_permute2f128_ps(xloVal, xhiVal, 0x20); // t0|t0|t1|t1|t2|t2|t3|t3
    b1Val = _mm256_permute2f128_ps(xloVal, xhiVal, 0x31); // t4|t4|t5|t5|t6|t6|t7|t7

    cVal = _mm256_fmadd_ps(a0Val, b0Val, cVal);
    cVal = _mm256_fmadd_ps(a1Val, b1Val, cVal);

    aPtr += 16;
    bPtr += 8;
  }

  __VOLK_ATTR_ALIGNED(32) float dotProductVector[8];

  _mm256_store_ps(dotProductVector, cVal); // Store the results back into the dot product vector

  *realpt = dotProductVector[0];
  *imagpt = dotProductVector[1];
  *realpt += dotProductVector[2];
  *imagpt += dotProductVector[3];
  *realpt += dotProductVector[4];
  *imagpt += dotProductVector[5];
  *realpt += dotProductVector[6];
  *imagpt += dotProductVector[7];

  number = eighthPoints*8;
  for(;number < num_points; number++){
    *realpt += ((*aPtr++) * (*bPtr));
    *imagpt += ((*aPtr++) * (*bPtr++));
  }

  *result = *(lv_32fc_t*)(&res[0]);
}
```

PFB benchmark (python)

```python
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: PFB Channelizer Perf
# Author: Arnaud Durand
##################################################

from gnuradio import blocks
from gnuradio import analog
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from gnuradio.filter import pfb
import filter_avx2
import argparse

GSM_RATE = 1625000.0 / 6.0
nchannels = 50

class top_block(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self, "Top Block")

        ##################################################
        # Variables
        ##################################################
        self.nchannels = nchannels
        self.transition_width = transition_width = 10e3
        self.samp_rate = samp_rate = GSM_RATE*nchannels
        self.gain = gain = 30.9
        self.cutoff_freq = cutoff_freq = 135e3
        self.pfb_taps = pfb_taps = firdes.low_pass(gain, samp_rate, cutoff_freq, transition_width, firdes.WIN_HANN)
        print len(pfb_taps)
        self.center_freq = center_freq = 0

        ##################################################
        # Blocks
        ##################################################
        self.pfb_channelizer_ccf_0 = filter_avx2.pfb_channelizer_ccf(
              nchannels,
              (pfb_taps), 1)
        #self.pfb_channelizer_ccf_0.set_channel_map(())

        self.blocks_null_sink_0 = blocks.null_sink(gr.sizeof_gr_complex*1)
        self.analog_sig_source_0 = analog.sig_source_c(samp_rate, analog.GR_COS_WAVE, 1000, 1)
        self.blocks_head_0 = blocks.head(gr.sizeof_gr_complex*1, int(samp_rate*60))

        self.s2ss = blocks.stream_to_streams(gr.sizeof_gr_complex*1, self.nchannels)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.analog_sig_source_0, 0), (self.blocks_head_0, 0))
        self.connect((self.blocks_head_0, 0), (self.s2ss, 0))
        for i in range(self.nchannels):
            self.connect((self.s2ss, i), (self.pfb_channelizer_ccf_0, i))
            self.connect((self.pfb_channelizer_ccf_0, i), (self.blocks_null_sink_0, i))


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("nchannels", help="benchmark PFB channelizer for nchannels",
                        type=int, default=50)
    args = parser.parse_args()
    nchannels = args.nchannels
    tb = top_block()
    import time
    start = time.time()
    tb.start()
    tb.wait()
    end = time.time()
    print "%.4gs" % (end-start)
```