

Enabling technologies for real-time acquisition and processing of large volumes of data and their applications to giant astronomical telescopes and radar systems

Julien Plante

▶ **To cite this version:**

Julien Plante. Enabling technologies for real-time acquisition and processing of large volumes of data and their applications to giant astronomical telescopes and radar systems. Astrophysics [astro-ph]. Université Paris sciences et lettres; Thales LAS France, 2023. English. ⟨NNT : 2023UPSLO015⟩. ⟨tel-04542844⟩

HAL Id: tel-04542844

https://theses.hal.science/tel-04542844v1

Submitted on 11 Apr 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



HAL Authorization

PSL UNIVERSITÉ PARIS

THALES

Région îledeFrance

THÈSE DE DOCTORAT

DE L'UNIVERSITÉ PSL

Préparée à l'Observatoire de Paris
Dans le cadre d'une cotutelle avec Thales LAS France

Technologies habilitantes pour l'acquisition et le traitement en temps-réel de gros volumes de données et leurs applications aux télescopes astronomiques géants et aux systèmes radars

Enabling technologies for real-time acquisition and processing of large volumes of data and their applications to giant astronomical telescopes and radar systems

Soutenue par
Julien PLANTE
Le 30/11/2023

Ecole doctorale n° 127
Astronomie et Astropysique d'Île-de-France

Spécialité
Astronomie
Traitement du Signal
Instrumentation

Composition du jury :

Françoise, COMBES	Profresseur, Observatoire de Paris	*Présidente*
Jason, HESSELS	Professor, University of Amsterdam	*Rapporteur*
Nicolas, GAC	Professeur, Université Paris-Saclay	*Rapporteur*
Chiara, FERRARI	Astronome, Observatoire de la Côte d'Azur	*Examinateur*
Béatrice, PESQUET-POPESCU	Professeur, Thales LAS	*Examinateur*
Damien, GRATADOUR	Chargé de recherche, ObservatoiredeParis	*Directeur de thèse*

Allocation Doctorale «Paris Region PhD» financée par la Région Île-de-France

Acknowledgements

I sincerely thank the French Region Île-de-France for funding this work through the Paris Region PhD program (DIM ACAV+ #20007183).

I am also deeply grateful for the support and mentorship of my thesis supervisor Dr Damien Gratadour and co-supervisor Lionel Matias. This work could not have been completed without their continuous support, connections, and deep knowledge and experience of the fields touched in this thesis.

I am especially grateful to Dr Florian Ferreira and Arnaud Sevin from Observatoire de Paris, for welcoming me into their team at pole HRA, and for the interesting interactions regarding Adaptive Optics, DPDK, and COSMIC.

I am also grateful to the Observatoire Radioastronomique de Nançay, especially Cédric Viou and Emmanuel Thétas for their help deploying this work on-site and Dr Louis Bondonneau for his counselling about FRB detection, as well as Dr Jean-Mathias Grießmeier (LPC2E, USN) and Dr Valentin Decoene (Nantes Université) for approving our piggybacking on the LT03 and LT05 observation programs. On a similar level, I thank Helène Menuet and Remi Kassab from the ARC unit of Thales LAS France, and François-Xavier Archambault for their help understanding radar system architecture and algorithms.

Many thanks as well to Stephen Jones (Nvidia), for his ever accurate insights about the intricacies of CUDA distilled during our meetings.

Finally, thanks should also go to Elena Agostini (Nvidia) for her implication in DPDK and her work to enable NIC to GPU DMA, as well as the collaboration we had about this matter.

On a more personal note, I deeply thank my family and my friends, especially my partner Margot, who endured this thesis as much as I did, and supported me flawlessly during these three long years.

I thank my friends and fellow PhD students Tomeu and Étienne for helping me vent, for the interesting discussions, and for pushing me until the last day of this thesis. I am grateful to my parents Vincent and Valérie for keeping interest in my long studies, and to both my grandparents Raphaël and Marie-Reine and my teachers Dr Anica Lekic and Pr Redamy Péres-Ramos for transmitting their inclination for research.

Finally, I want to name every other significant close family member or friend, who all contributed to support me during this work: my grandmother Raymonde, my aunts and uncles Christelle, Aline, Christophe and Stéphane, my cousins Coralie, Lorina and Émeline, my family-in-law Denis, Delphine, Carla and Mattéo, my friends Victor, Yves, Alexandre and Marc and my dog Tiplo.

Besides the technical and scientific aspects, this work was also a strong human experience, and would not have been possible without them.

Abstract

Systems of our world are following a constant growth in terms of precision, speed and electrical consumption. This growth involves the production of always larger volumes of data, which become difficult to process using today's standard technologies.

In the context of this thesis, we looked into emerging technologies such as GPU computing, userland networking, DPDK and GPUDirect, which are able to answer this need, and applied them to multiple real-life projects with strong scientific and industrial impact (wavefront acquisition for adaptive optics, real-time fast radio burst detection, radar signal processing) with success.

As a first application, high-performance wavefront acquisition is a corner stone of Adaptive Optics systems, essential to the quality of images produced by ground-based telescopes. On the ELT, the Adaptive Optics control loop must not be longer than 200 µs, although network packet acquisition with standard methods on the Linux OS already exhibits a latency of multiple tens of microseconds.

A second application is the detection of elusive signals sharing similarities with pulsars, the Fast Radio Bursts. The cutting-edge technologies considered in this thesis enable real-time detection of these signals at very low frequency on the telescope NenuFAR, making it possible to search for them continuously, statistically increasing chances of detection.

Finally, a third application is the acquisition and processing of radar data, providing technical solutions to the current performance bottlenecks encountered in this domain. This opens way to better resolution and sensitivity.

We show how these technologies converged to reusable solutions across our multiple applications, and provide performance analyses. We obtain encouraging results, providing a solution to the current bottleneck of current standard technologies. This is promising in the context of the extreme systems being designed in the domains considered (SKA, giant radars), and could also be transposed to other domains (autonomous vehicles, finance, *etc*).

Résumé

Les systèmes de notre monde sont en croissance constante en termes de précision, rapidité et consommation énergetique. Cette croissance passe par la production de volumes de données toujours plus importants, qui deviennent difficiles à traiter avec les technologies standard actuelles.

Dans le cadre de cette thèse, nous nous sommes intéressé à des technologies émergentes, telles que le calcul sur GPU, le réseau en espace utilisateur, DPDK et GPUDirect, permettant de répondre à ce besoin, et les avons appliquées à plusieurs projets concrets à fort impact scientifique et industriel (acquisition de front d'ondes pour optique adaptative, detection en temps réel de transients radio, traitement du signal radar) avec succès.

La première application est l'acquisition à haute performance de fronts d'ondes, une pierre angulaire de l'Optique Adaptative, essentielle à la qualité des images produites par les telescopes terrestres. Pour l'ELT, la boucle de contrôle d'Optique Adaptative ne doit pas dépasser 200 µs, alors que l'acquisition de paquets réseau à partir du système d'exploitation Linux possède une latence de plusieurs dizaines de microsecondes.

La deuxième application est la détection de signaux encore peu compris, possédant des caractéristiques similaires aux pulsars, les Sursauts Radio Rapides. Les technologies de pointe étudiés dans cette thèse permettent la détection en temps-réel de ces signaux à très basse fréquence sur le téléscope NenuFAR, rendant possible une recherche continue de ces signaux, augmentant statistiquement les chances de détection.

Enfin, la troisième application concernce l'acquisition et le traitement de données radar, en proposant des solutions techniques aux limites de performance rencontrées dans ce domaine. Cela ouvre la voie à une meilleure résolution et sensibilité.

Nous montrons comment ces technologies ont pu converger vers des solutions réutilisables dans le contexte de ces différentes applications, et fournissons des analyses de performance. Nous obtenons des résultats encourageants, proposant ainsi une solution aux limitations des technologies standard actuelles. Ces résultats sont prometteurs dans le cadre des systèmes extrêmes en court de conception dans les domaines considérés (SKA, radars géants), et pourraient également être transposés à d'autres domaines (véhicules autonomes, finance, *etc*).

Contents

Acknowledgements **2**

Abstract **3**

Résumé **3**

Glossary **7**

Abbreviations **9**

Symbols **14**

Introduction **16**

I High-performance data acquisition **18**

Introduction **19**
Previous work . . . 19
Problem statement . . . 20

1 Data acquisition system description **22**
1.1 Architecture . . . 22
1.1.1 Experiment configuration . . . 22
1.1.2 Host packet processing, Linux kernel RX . . . 23
1.1.3 Host packet processing, DPDK RX . . . 23
1.1.4 GPU packet processing, DPDK . . . 25
1.1.5 GPU packet processing, DPDK gpudev . . . 26
1.1.6 Packet processing persistent kernel design . . . 26
1.2 Performance tuning . . . 27
1.2.1 PCIe topology . . . 28
1.2.2 PCIe usage . . . 28
1.2.3 NUMA effects . . . 29
1.2.4 Burst size . . . 29
1.3 Telemetry . . . 30
1.3.1 Data dumping . . . 30
1.3.2 Pipeline control and monitoring . . . 31

2 Applications **33**
2.1 Common network protocols . . . 33
2.1.1 Endianness . . . 34
2.1.2 Ethernet . . . 34
2.1.3 IPv4 . . . 34
2.1.4 UDP . . . 35
2.1.5 Note on alignment . . . 36

2.2 Adaptive Optics . . . 37
2.2.1 ESO network protocols . . . 38
2.2.2 ELT-MICADO . . . 39
2.2.3 VLT-MAVIS . . . 41
2.2.4 Results . . . 41
2.3 Radio astronomy . . . 43
2.3.1 BIGCAT . . . 44
2.3.2 NenuFAR . . . 45
2.4 Radar . . . 47
2.4.1 Protocol . . . 47
2.4.2 Acquisition system . . . 48
2.4.3 Results . . . 49

3 Future work **50**
3.1 Portability . . . 50
3.1.1 Other NIC + Nvidia GPU . . . 50
3.1.2 Nvidia NIC + other GPU . . . 50
3.1.3 Other NIC + other GPU . . . 50
3.2 Alternatives to DPDK . . . 50
3.2.1 DPU . . . 51
3.2.2 GPUNetIO . . . 51
3.2.3 Newer functionalities . . . 51
3.2.4 RDMA . . . 52
3.3 Encapsulation in a high-level component . . . 53

Conclusion **54**

II High-performance GPU computing **55**

Introduction **56**

4 Methodology **57**
4.1 CUDA kernel optimization . . . 57
4.2 Benchmarking with many degrees of freedom . . . 58
4.2.1 Mathematical models . . . 58
4.2.2 GPU saturation . . . 58
4.2.3 Uncertainty . . . 58

5 Radar **60**
5.1 Porting an existing CPU-based radar SP . . . 60
5.1.1 A secondary radar . . . 60
5.1.2 Existing SP implementation . . . 62
5.1.3 Implementation . . . 63
5.1.4 Results . . . 64
5.1.5 Future work / Lessons learned . . . 65
5.2 Primary radar . . . 66
5.2.1 Classical primary radar SP . . . 66
5.2.2 Increasing the number of hypotheses . . . 68
5.2.3 Implementation: `ConvSP` . . . 70
5.2.4 PC implementation . . . 78
5.2.5 DF implementation . . . 78
5.2.6 LogMod . . . 86
5.2.7 CFARs . . . 86
5.2.8 Full pipeline . . . 87
5.2.9 Kernel fusion . . . 87
5.2.10 Future work . . . 89

6 FRB detection on NenuFAR **91**

7 Interesting connections between radar and radioastronomy **124**
7.1 Beamforming . . . 124
7.2 Feature extraction . . . 125
7.3 Adaptive thresholding . . . 125

8 Future work **127**
8.1 Alternatives to CUDA . . . 127
8.1.1 OpenCL . . . 127
8.1.2 ROCm / HIP . . . 128
8.1.3 SYCL . . . 128
8.1.4 Others . . . 128
8.1.5 Going to a higher level . . . 128
8.2 Multi-GPU, multi-node systems . . . 129
8.3 COSMIC framework . . . 130

Conclusion **131**

Conclusion **134**

Code availability **136**

A Papers **141**

B Parset file **156**

C Details on GPU architecture **158**
C.1 GPU memory subunits . . . 158
C.1.1 global memory . . . 158
C.1.2 local memory . . . 158
C.1.3 shared memory . . . 158
C.1.4 constant memory . . . 159
C.1.5 texture memory . . . 159
C.1.6 surface memory . . . 159
C.1.7 Other . . . 159
C.2 Compute Capability . . . 160
C.3 Coalesced memory accesses . . . 160

Glossary

ALICE, **LISA**, **FREDA** } **WFS** Different WaveFront Sensors (WFSs) of MICADO

alignment Property of a computer address to be a multiple of some number, commonly a power of 2. For example, the address `0x1234` is aligned to 4 B, `0x1230` is aligned to 16 B, but `0x4321` is not aligned

beamlet

CUDA block Group of threads

CUDA kernel Function dispatched over multiple CUDA CUDA blocks and threads, running in parallel on an Nvidia Graphics Processing Unit (GPU)

CuTe Sister library of CUTLASS providing an elaborate `Tensor` class to describe n-D arrays of data with a lot of flexibility

CUTLASS High-performance, header-only linear algebra from Nvidia, with the main goal of targeting Tensor Cores

device External system, controlled by the host. In the context of CUDA, this is a GPU.

DOCA

global, **local**, **shared**, **constant**, **texture**, **surface** } **memory** Different memory subunits related to the GPU. A detailed description is given in C.1

goodput Good + throughput – Payload bytes received over a certain period of time, stripped from network headers

gpudev DPDK library to use GPUs in an abstracted way.

host Orchestrator of a computer, typically the CPU

io_uring Asynchronous interface to Linux I/O

jumbogram L4 packet with a payload greater than MTU

kernel fusion Action of fusing multiple CUDA kernels into one, in order to improve performance.

Linux kernel Core of the Linux OS, handling tasks such as multiprocessing, memory management, device management, etc

NCHW, **NHWC**, **KCRS** } **layout** Different memory layouts

sensitivity Proportion of true positives among all positive detections

specificity Proportion of true negatives among all negative detections

system call Function to request a service from the OS, including I/O, process creation, scheduling, ...

thread Thread of execution, a succession of instructions

Abbreviations

ADC Analog-to-Digital Converter

ADS-B Automatic Dependent Surveillance-Broadcast

AI Artificial Intelligence

AO Adaptive Optics

API Application Programming Interface

ASIC Application Specific Integrated Circuit

ATCA Australian Telescope Compact Array

AVX Advanced Vector eXtensions

BHR Beamformed High Resolution

BIGCAT Broadband Integrated Gpu Correlator for ATCA

BLAS Basic Linear Algebra Subprograms

BU Business Unit

C2C Complex-to-Complex

CA-CFAR Cell Averaging CFAR

CC Compute Capability

CCD Charge-Coupled Device

CFAR Constant False Alarm Rate

CNN Convolutional Neural Network

COTS Commercial off-the-shelf

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CSIRO Commonwealth Scientific and Industrial Research Organisation

CSP Central Signal Processor

cuBLAS CUDA BLAS

CUDA Compute Unified Device Architecture

cuDNN CUDA DNN

cuFFT CUDA FFT

cuFFTDx cuFFT Device eXtension

DCCP Datagram Congestion Control Protocol

DCS Double Correlated Sampling

DF Doppler Filtering

DGX Deep learning GPU eXtreme

DMA Direct Memory Access

DMT Dispersion Measure Transform

DNN Deep Neural Network

DPC++ Data Parallel C++

DPDK Data Plane Development Kit

DPU Data Processing Unit

DRAM Dynamic RAM

DSCP Differentiated Services Code Point

dst destination

eBPF extended Berkeley Packet Filter

ECN Explicit Congestion Notification

ELT Extremely Large Telescope

EoIB Ethernet over InfiniBand

EPI European Processor Initiative

ESA European Spatial Agency

ESO European Southern Observatory

FAD Functional Analysis Document

FDMT Fast Dispersion Measure Transform

FFT Fast Fourier Transform

FIFO First In, First Out

FLOP Floating Point OPeration

FMA Fused Multiply Add

FPGA Field-programmable Gate Array

FRB Fast Radio Burst

FRUIT False Replies Unsynchronized with Interrogator Transmissions

FS Fowler Sampling

GDDR Graphics Double Data Rate

GDS GPUDirect Storage

GEMM GEneral Matrix Multiply: $D = \alpha AB + \beta C$

GNSS Global Navigation Satellite Systems

GPGPU General Purpose GPU

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HDF Doppler Filter Hypothesis

HPC High Performance Computing

HRA Haute Résolution Angulaire (High Angular Resolution)

I/O Input/Output

IB InfiniBand

IHL Internet Header Length

IP Internet Protocol

IPoIB Internet Protocol over InfiniBand

IPsec Internet Protocol Security

IPv4 IP version 4

IPv6 IP version 6

IQ In phase, Quadrature

I²C Inter-Integrated Circuit

JATIS Journal of Astronomical Telescopes, Instruments and Systems

JSON JavaScript Object Notation

LAN Local Area Network

LESIA Laboratoire d'Études Spatiales et d'Instrumentation en Astrophysique (Space and Astrophysics Instrumentation Research Laboratory)

LOFAR Low-Frequency Array

LogMod Logarithm of the Modulus

LSB Least Significant Byte

MAC Media Access Control

MAD Median Absolute Deviation

MAVIS MCAO Assisted Visible Imager and Spectrograph

`mbuf` memory buffer

MCAO Multi-Conjugate AO

MICADO Multi-AO Imaging Camera for Deep Observations

MMA Matrix Multiply Accumulate

MMHRSP Massively Multi-Hypotheses Radar SP

MPPA Massively Parallel Processor Array

MSB Most Significant Byte

MTU Maximum Transmission Unit

MUDPI Multicast UDP Interface

NenuFAR New Extension in Nançay Upgrading LOFAR

NIC Network Interface Controller

NUMA Non-Uniform Memory Access

NVMe Non-Volatile Memory Express

OS Operating System

OS-CFAR Ordered-Statistic CFAR

OSI model Open Systems Interconnection model

PC Pulse Compression

PCI Peripheral Component Interconnect

PCIe PCI Express

PCIe Gen PCIe Generation

PDCA Plan, Do, Act, Check

POSIX Portable Operating System System Interface

PTP Precision Time Protocol

radar RAdio Detection And Ranging

RAM Random Access Memory

RDMA Remote DMA

regex Regular Expression

RFI Radio Frequency Interference

RLE Run-Length Encoding

RoCE RDMA over Converged Ethernet

RTC Real-Time Computer

RTMS Real-Time MUDPI Stream

RX Reception

SCTP Stream Control Transmission Protocol

SDP Science Data Processor

SIMD Single Instruction, Multiple Data

SKA Square Kilometer Array

SM Streaming Multiprocessor

SNR Signal-to-Noise Ratio

SP Signal Processing

SR Single Read

SRP Simple Radar Protocol

SSE Streaming SIMD Extensions

SSR Secondary Surveillance Radar

TCP Transmission Control Protocol

TMA Tensor Memory Accelerator

ToS Type of Service

TTL Time To Live

TUI Terminal User Interface

TX Transmission

UDP User Datagram Protocol

UML Unified Modelling Language

USB Universal Serial Bus

vDPA virtio Data Path Acceleration

VLAN Virtual LAN

VLT Very Large Telescope

VPP Video Processing Pipeline

VRX Video RX

WFS WaveFront Sensor

WGMMA WarpGroup MMA

WIP Work-in-Progress

WMMA Warp MMA

XDP eXpress Data Path

Yacc Yet Another Compiler-Compiler

Symbols

$\lceil . \rceil$ Ceil function

$\lfloor . \rfloor$ Floor function

$\lfloor . \rceil$ Round to nearest integer

$[\![a, b]\!]$ Integer range from a to b, both inclusive

f_s Sampling frequency

$\Im(z)$ Imaginary part of z

$\Re(z)$ Real part of z

Introduction

Cyberphysical systems, that can be described as systems interacting with the real world by processing data from sensors, surround us more than ever in the present world[1, 65, 28]. Autonomous vehicles, cellphones, and smart factories are among the most famous and common examples of such systems. Following Moore's law, the amount of data received and processed by these systems has been steadily growing over the last decades, generating many scientific and industrial revolutions at the same time[62, 7], in order to keep up with this exponential performance growth. In this thesis, we focused on the case of two extreme cyberphysical systems, featuring some of the largest data rates and most severe operational constraints in the world: giant astronomical telescopes and radar systems.

By focusing on these two classes of systems, we identified the start of a new revolution regarding data acquisition, and an ongoing one with regards to data processing.

A majority of data transfers in cyberphysical systems is based on standard networking protocols, built upon the Ethernet protocol[45]. For a long time, corresponding data acquisition systems have been relying on Operating System (OS) features in order to handle interface with the actual receiving hardware, commonly Network Interface Controllers (NICs), but also in some rarer cases wireless LAN controllers or baseband devices. However, after decades of performance being limited by hardware, data acquisition has recently become limited by software, because of the architecture of current OSes[36]. Complex solutions based on Field-programmable Gate Arrays (FPGAs) have been designed in order to lift this limitation[52, 53], but recent developments of Smart NIC devices caused a small revolution in the data acquisition domain. In this thesis, we investigated the use of such new technology in order to provide a solution for high-performance data acquisition in cyberphysical systems based on Commercial off-the-shelf (COTS) hardware.

The second, ongoing revolution in the context of High Performance Computing (HPC) is the progressive adoption of accelerators or co-processors such as GPUs, featuring much denser computational power than the regular Central Processing Units (CPUs)[29]. Starting in the 2000s, and soaring with the advent of Artificial Intelligence (AI)[5], GPUs feature an massively parallel architectures, able to process a number of threads of execution three orders of magnitude larger than with CPUs. This is made possible by greatly simplifying the execution units compared to a CPU, and enables computational intensities two to three orders of magnitude larger than with CPUs on appropriate algorithms. Indeed, the parallel architecture and reduced instruction set of the GPU makes it interesting only for the most repetitive and large scale use cases. This matches our need in extreme cyberphysical systems, and motivated the use of GPU computing as a means to process the large data rates acquired in this context.

This thesis is dedicated to providing a better understanding of these two technological revolutions, and promoting their use in every cyberphysical system.

In order to do so, we focused on three main applications: Adaptive Optics, Fast Radio Burst detection, and radar Signal Processing.

Adaptive Optics (AO) is a technology used in optical systems to correct aberrations caused by a transmitting medium[15, 58]. The main use of AO is ground-based astronomy, in order to compensate for the effects of the atmosphere and produce clean images of stars and other astronomical objects using ground-based telescopes. This is especially key to the discovery of exoplanets, and the study of these distant worlds. Moreover, this technology has been applied with similar goals in the medical domain, for retinal imaging[38] and microscopy[42]. In the context of this thesis, we focused on the use of AO in astronomy.

The principle of AO is to measure aberrations on a wavefront using a WFS, and to use a deformable mirror to compensate for them. Computations are required between these two stages, and are performed by a Real-Time Computer (RTC), forming a control loop[58]. Because the effects of the atmosphere can change very rapidly, this control loop must maintain a very low latency, typically under 200 µs, making it qualified as an extreme cyberphysical system. Since the first AO systems for astronomy deployed in the late 1980s, WaveFront Sensors have greatly increased in resolution, and deformable mirrors have gained in number of actuators, making complexity increasing quadratically. This makes this application a prime candidate for experimenting cutting edge technologies. Moreover, the AO team of LESIA has been developing GPU-based AO RTCs for over a decade, and is looking into newer data acquisition methods in order to benefit from the latest hardware and sensors.

A second application considered in the context of this thesis is radio astronomy, and especially the problem of Fast Radio Burst (FRB) detection in real-time. The study of FRBs is a nascent research topic in radio astronomy, since the first of these elusive signals was detected in 2007[39]. Since then, a few thousands of these bright, intense, and isolated events have been detected, but hundreds of them could theoretically be received every day. Limitations for detection of these events includes Radio Frequency Interference (RFI), mainly caused by human activity, low Signal-to-Noise Ratio (SNR), and very high memory footprint[19]. Indeed, the signal of FRBs is dispersed during its propagation through the interstellar medium, and its lower frequency components are received with a delay compared to higher frequencies. This delay is quadratically inverse to the observation frequency, and can range from a few milliseconds to multiple hours. Because of this, only a small portion of the existing FRBs are actually detected. Relative lack of data, especially in the lower observing frequencies, make them a challenging phenomenon, and their emission mechanism is not yet understood, even though models have been proposed.

In the context of this thesis, we tried to provide a technical solution to the problem of FRB detection at lower frequencies, by proposing a real-time FRB detection pipeline for the telescope NenuFAR. This telescope, built in Nançay, France and extending the European telescope Low-Frequency Array (LOFAR), is currently one of the most powerful (sensitivity, resolution) in the very low-frequency range (10–85 MHz). It has not detected any FRB at the time of writing, making it a very interesting challenge. In the current approach used on New Extension in Nançay Upgrading LOFAR (NenuFAR), large campaigns of observation are made, and data is stored on disk, after some reduction operations (time integration and Short-time Fourier Transform). The resulting data is then processed, in hopes of finding FRBs. This approach provides the advantage of enabling very good observability and reproducibility, but suffers from a huge storage footprint. This limits greatly the amount of data in which FRBs can be searched. Moreover, time integration induces a loss of information that can reduce SNR, and as such reduce the probability of detection.

We propose a real-time FRB detection pipeline for NenuFAR, in order to address the limitations of the current method and increase the probability of detection of these interesting signals. The proposed pipeline has the dual goal of providing interesting scientific results on NenuFAR, and additionally to demonstrate the use of the enabling technologies of GPU computing and high performance networking in the context of radio astronomy. This is motivated by the upcoming giant radio telescope Square Kilometer Array (SKA), of which NenuFAR is an official pathfinder.

Finally, the third domain of application considered is that of radar systems, where both primary and secondary radars were considered. Primary radars systems are used for airspace surveillance without cooperation of aircraft[56]: a signal is sent, and potential aircraft are detected by their echo to this initial impulsion. No action is required from the detected aircraft, making it possible to detect any target (planes, missiles, drones, or even rain or birds). Secondary radars, on the contrary, are used for airspace surveillance with collaborative targets[22], and complement primary radars. This second type of radar interrogates an aircraft's transponder, giving various information such as identification, altitude, or position.

In the context of this thesis, we focused on porting an existing secondary radar Signal Processing application from CPU to GPU, and on the development of a scaled-up Signal Processing chain benefiting from GPU acceleration for primary radars, opening great perspectives for future radar computing systems, as well as linking this pipeline with our high performance data acquisition system.

In the first part of this thesis, we will present the high performance data acquisition system based on COTS hardware that was developed throughout this work. We start by explaining its global architecture in Chapter 1, proceed by presenting how this system was applied to our different applications in Chapter 2, then list potential future improvements in Chapter 3.

The second part of this thesis is related to GPU computing, and how this cutting-edge technology was applied in the two major applications of this work. We begin by presenting a set of common methodologies used throughout this thesis in Chapter 4, then go on with our two applications: radar systems in Chapter 5 and astronomy, through FRB detection, in Chapter 6. Finally, we expose the non-trivial connections between these two applications, and the associated opportunities it creates in Chapter 7, and future improvements for these different GPU computing applications in Chapter 8.

Part I

High-performance data acquisition

Introduction

Throughout the last few decades, the maximum networking data rate enabled by Ethernet hardware roughly followed Moore's law, doubling every two years. Until recently, the main data rate bottleneck for acquisition interfaces in cyberphysical systems was coming from the hardware, namely NICs and Ethernet links.

However, during the last decade, with the popularization of 10, 40 and 100 GbE hardware, the bottleneck shifted from hardware to software. Indeed, the commonly used OS networking stacks suffer from lack of support for asynchronous operations, and introduce some overheads that were not limiting at lower data rates, but became impactful starting from 10 Gbit/s.

In the mean time, computing power available in cyberphysical systems skyrocketed, benefitting from the rise of AI and new accelerators making it possible, especially GPUs. This new type of processor unlocked new possibilities in term of computations while keeping costs and energy consumption low, which made them ubiquitous in modern systems.

In the context of AI, data is commonly stored on disk or in Random Access Memory (RAM), and accessed quickly. However, in cyberphysical systems, data acquisition can become a strongly limiting factor, especially in the context of the limitations of common networking stacks mentioned earlier.

In this part of the thesis, we will present the method we developed to create a very high performance data acquisition system, acquiring data from an Ethernet network and storing the acquired data in the GPU's Dynamic RAM (DRAM). This system was key to many applications considered in this thesis, and represents a strong contribution that could be reused in many other cyberphysical systems.

Previous work

Such data acquisition system is not a new need, but comes as the continuation of many systems that proved to be hardly scalable to newer hardware, namely the combination of $\geq$10 GbE NICs and GPUs. We will start by giving an overview of related efforts.

Linux kernel networking stack based acquisition

The most significant share of systems relying on high-speed networking are based on the Linux OS, and as such we will focus on this platform during the rest of this thesis. The current mainstream solution used to take fully advantage of newer NICs relies on heavy tuning of the Linux kernel[36]. By carefully managing the NIC's number of Reception (RX) queues, packet sizes and interrupts, as well as using vectorized packet reception calls and multithreading, it was shown to be possible to reach 10 Gbit/s using only the Linux networking stack.

However, fully dedicating multiple CPU cores to data acquisition is neither scalable nor resource efficient[35, 34]. Moreover, acquiring data through the Linux networking stack makes it possible to receive data in host memory, accessible only from CPUs. Because of this, an additional transfer is required to make GPU computing possible, adding yet another overhead.

Because of its standard aspect, this method has been used in Thales radars since Ethernet was chosen to transfer data from antennas to Signal Processing (SP) chains. It has also been used successfully in a number of projects in radio astronomy[66, 21, 64], and others[43, 61, 44].

FPGA based acquisition

One alternative to the standard Linux networking stack found to overcome its limitations is the use of FPGA-based data acquisition systems. Because of their much more flexible nature, FPGAs have been used to provide similar functionalities to NICs, although with specialized behavior in order to offload a maximum number of tasks from the Linux kernel to the FPGA itself, enabling the acquisition of very high data rates[53, 27].

In the case of the Green Flash project[1], coordinated by LESIA at Observatoire de Paris for AO, this was successfully used and deployed, even providing Direct Memory Access (DMA) functionalities to a GPU[53], effectively providing best possible performance.

However such approach is hardly reusable from one project to another, as FPGA IP blocks must be significantly rewritten for each different protocol supported, and configured for each different machine architecture. This leads to a very complex maintenance strategy, and the acquisition system proposed in the context of this thesis aims to provide easier maintenance and portability.

DPDK based acquisition

Another relatively widespread alternative to the Linux networking stack is the Data Plane Development Kit (DPDK) library. This library makes it possible to replace the Linux kernel for networking operations and to control directly the NIC's driver. This has already been used for various deployed projects, especially in radio astronomy[2, 10, 12], and has been used to receive bandwidths of several tens of gigabits per second on regular NICs.

However, DPDK was never used in previous applications coupled with GPU, and would still require in such cases an additional memory copy stage to a GPU, adding an additional overhead.

Interestingly, coincidentally to the thesis, Nvidia introduced in DPDK the possibility to perform NIC $\rightarrow$ GPU DMA, effectively lifting the mentioned limitation, with 5G networking as a main target. This made this state-of-the-art technology a very interesting candidate for implementation of a high-speed data acquisition system.

Problem statement

Because of existing GPU computing pipelines developed at LESIA, this data acquisition system needs to store received data in GPU memory, ready to be processed by existing pipelines.

We drew up a list of desirable features in Table 1, as well as the applications it applies to the most among those considered in this work. Note that even if SKA is a radio telescope, its dimensions are several orders of magnitude larger than that of NenuFAR, hence some features were classified as desirable only for SKA, and some for radio telescopes in a more broader sense.

In the following chapters of this part of the thesis, we will describe the data acquisition system that we designed to best answer these requirements. Then, we proceed by describing how this system was applied to the multiple domains we considered. Finally, possible future improvements are discussed.

[1] https://greenflash-h2020.eu/

Feature	Description	Application
Reliability	The system must be able to work during multiple weeks without any packet loss or other error	All
Maintainability	The system must be maintainable during multiple decades, both in terms of hardware (issue of accessibility to a specific model) and software (issue of code complexity, clarity and documentation)	All
Low latency	The system must introduce as little latency as possible	AO+++ radar++
High bandwidth	The system must be able to perform the acquisition of large volumes of data	SKA+++ radar+++ Others
Self-contained	The system must work in itself, not relying on a modification of the whole emission chain we are acquiring data from	Radio astronomy+++ AO+++ radar+
Scalability	The system should be able to scale to larger problems	All
Reusability	The system should be reusable, with the goal to provide a generic high-level tool for high-performance data acquisition	All
Energy efficiency	The system should be as energy efficient as possible both for ecological and infrastructure (maximum energy consumption and thermal dissipation) concerns	SKA+++ radar++ Others

Table 1: Feature requirements for data acquisition method

Chapter 1

Data acquisition system description

In this chapter, we explain how we used the DPDK framework in order to design a solution to the problem described in Section I, and why was this framework used compared to other solutions.

We start by describing the architecture of the resulting data acquisition system, continue by giving a performance tuning guide for this system in order to obtain the best performance on a given hardware topology, and finish by expanding on the telemetry method used to inspect and debug the states of systems built with this data acquisition system.

1.1 Architecture

In this section, we will list the different iterations that were developed before stabilizing our approach to high-performance data acquisition, as well as the final designs. We will discuss the advantages of each approach, and show why it is difficult to unify bursted and continuous acquisition schemes.

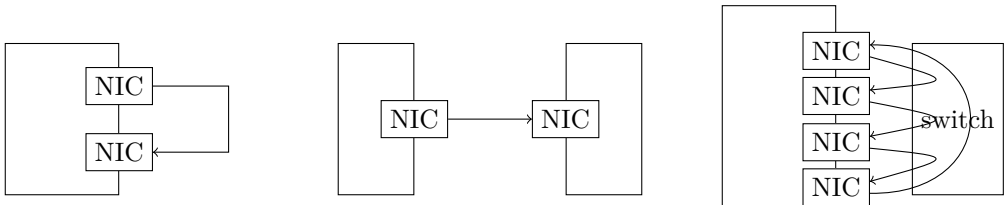
1.1.1 Experiment configuration

In order to develop, test and benchmark the different approaches developed around high performance data acquisition, we developed multiple experiments. Two reasons motivated the existence of multiple different configurations:

- Privacy policy: Because of the confidential nature of a part of Thales' activity, many limiting rules exist regarding network access, code sharing, software installation, *etc.* As a result, experiments related to Thales had to be realized on site at Limours, on Thales machines, and code could not be shared between Thales and the Observatoire de Paris. This lead to a separate experiment configuration, and a separate codebase.
- Metrics: Latency is challenging to measure as explained in paragraph 2.2.4.1.1. The loopback configuration makes it much easier to measure latency, while being less realistic.
- Realism: Metrics collection is important, but testing in conditions as close to reality as possible was another important task in order to remove the possible bias of a simplified configuration, thus improving confidence in our data acquisition system.

Figure 1.1 presents in more details three major configurations used. The loopback configuration, where two NICs of the same machine are connected together, enables easier time management and latency computation, as no specific clock synchronization has to be performed, although care must be taken in case the hosting machine possesses multiple hardware clocks. The point to point configuration, where two NICs of different machines are connected, is the most representative of a real-life system. Finally, the switched loopback configuration is a variant of the simple loopback

configuration, featuring more NICs connected through a network switch. This configuration was especially useful to try and scale up 100 GbE to higher bandwidths by distributing acquisition among multiple NICs. This was additionally a configuration pushing the limits of the PCI Express (PCIe) bus, because of the full-duplex communication scheme used.



(a) Loopback (b) Point to Point (c) Switched loopback

Figure 1.1: Overview of experiment configurations

1.1.2 Host packet processing, Linux kernel RX

From initial testing using the `iperf3` tool and custom code, User Datagram Protocol (UDP) transfer fails to overcome a limit of 4 Gbit/s when using standard Linux kernel Application Programming Interfaces (APIs):

- `send` / `recv`: send / receive a single packet
- `sendmsg` / `recvmsg`: send / receive a single packet with more control
- `sendmmsg` / `recvmmsg`: send / receive multiple packets with same control capabilities as `sendmsg` / `recvmsg`.

Even with the latter option which provides vectorization, and non-blocking sockets, we failed to achieve more than 10 Gbit/s. This is explained by the cost of the Linux context switches[16, 35]: when calling a system call, the current program is paused, and replaced temporarily by the kernel context, before being resumed. This operation has a significant overhead, and can become a bottleneck if too frequent.

1.1.3 Host packet processing, DPDK RX

An alternative to the Linux kernel that can be used to reduce the number of context switches and obtain better performance is userland libraries. As their name suggests, userland libraries execute code from user space, as opposed to kernel space. The distinction is generally used in contexts where a library replaces OS functionalities, such as networking in the case considered here, by functions usable from user space. One of the most widespread such libraries for networking is DPDK. In this section, we will begin by presenting this library in more details, and describe the main steps of setting up an acquisition system using it.

1.1.3.1 The Data Plane Development Kit

The DPDK library was first released in 2010 by Intel, and is being developed and maintained as one of the Linux Foundation Projects.

It provides a de-facto standard interface to NIC drivers, abstracting the many different drivers available, usually handled by the Linux kernel. As of version 23.07, it supports 58 different networking drivers, including 9 *meta*-drivers. We designate by *meta*-drivers those controlling virtual devices, such as a classic Linux kernel socket, files, or shared memory. By writing software with DPDK, we can target any of these drivers, albeit with a varying level of success, since not all features of DPDK are supported by every driver[14].

It is worth noting that over time, DPDK accumulated abstractions for hardware other than NICs: baseband devices, crypto devices, compression devices, virtio Data Path Acceleration

(vDPA) devices, Regular Expression (regex) devices, machine learning devices, DMA devices, GPU devices and NICs supporting an event-driven scheme of operation. This gradually increased the complexity of the library but makes it suitable for many use cases. However, this choice is debatable, since some other abstractions, especially for GPU, may be more popular (see Section 8.1), resulting in an unnecessary abstraction effort.

1.1.3.2 Typical setup of an acquisition system based on DPDK

The typical skeleton of a DPDK application is given below, using jargon explained in greater depth in the continuation of the section.

- `rte_eal_init`: Initialize DPDK
- `rte_pktmbuf_mpool_create`: Allocate memory pool(s)
- `rte_eth_dev_start`: Start receiving packets on a NIC
- in a loop `rte_eth_rx_burst`: Transfer burst of packets from a NIC to a memory pool
- (cleanup): Free memory, close NIC(s), …

1.1.3.2.1 Initialization Every program relying on DPDK must initialize the library early in the application, through `rte_eal_init`. This function scans the server it runs on in search for any device supported by DPDK. Note that the library was primarily designed to scan the PCIe, but also support the virtual devices presented earlier.

Hugepage availability is checked at this stage too. Indeed, in order to provide best-in-class performance, DPDK relies on hugepages, a feature of modern OSes used to reduce the time spent managing virtual memory: using larger memory chunks (the huge pages), less different pages have to be managed, and operations are faster. DPDK handles this additional complexity automatically through its custom memory management primitives.

These memory management primitives are generally hidden by another level of abstraction, the memory pools. These data structures provide a ring of memory zones, allocated once and managed by DPDK. These memory zones can be acquired and released, in a similar fashion than when allocating and freeing memory, but without actual additional allocation, and thus with a very minimal performance impact. Because of this desirable feature, packets in DPDK are transferred from NIC to memory pool.

More precisely, one memory pool is attached to every RX queue allocated for a NIC. Different NICs support varying numbers of these RX queues, which makes it possible to receive multiple different streams of data on the same NIC easily. Note that on lower-end NICs, using multiple RX queues can also be mandatory in order to reach maximum performance. The best practice regarding this point is to allocate the preferred number of RX queues suggested by the NIC itself, that can be accessed through the field `default_rxportconf` of the `rte_eth_dev_info` structure.

Once all RX queues have been configured using `rte_eth_rx_queue_setup`, the NIC can be started, after which points ingress packets start being received instead of dropped. This is done with the function `rte_eth_dev_start`.

The final step of initialization concerns DPDK's Flow API. This is an advanced mechanism enabling to take specific actions on ingress and egress traffic, based on a concept similar to P4's match-action tables. Indeed, a set of rules is built to filter ingress traffic, and take action in case of match (enqueue in a specific RX queue, increment a counter, drop the packet, …). This feature is very powerful, and made it possible to implement robust systems with limited effort. Indeed, network programming using DPDK implies a full rewrite of the networking stack, which is a significant task. Using the Flow API, we can discard packets that do not match our rules, which can at least result in a packet to be dropped, and at best to be redirected to the Linux kernel for standard processing. This is noticeably the case with Nvidia NICs and the MLX5 driver from Nvidia (formerly Mellanox), that we used throughout this work.

1.1.3.2.2 Main loop The main loop of a data acquisition system based on DPDK with packet processing on the host can be broken down in four simple steps:

1. Receive packets with `rte_eth_rx_burst`.
2. Parse headers, and follow the networking protocols
3. (use the received data, *e.g.* copy to a managed location or process in-place)
4. Cleanup, free the memory associated to the packets received

Note that we put the third step into brackets as it is not a part of the data acquisition system *per se*, but is an important part (if not the most) in a real system.

We will proceed by giving important details on these steps. As explained in the previous paragraph regarding initialization, memory pools are associated with RX queues. When receiving packets in the first step of the main loop, memory buffers (`mbuf`s) are allocated from the associated memory pool, and filled with a number of packets. One `mbuf` contains one packet, plus additional metadata, most importantly linking to the next `mbuf` in case of fragmentation. Other metadata is available, in some cases strongly depending on the specific hardware and driver used, such as for timestamps.

The second and third steps are dependant on the specific application targeted. In this specific scheme, the only fixed design is that they are executed by a CPU.

Finally, the last step releases the `mbuf` acquired at step 1. This memory segment becomes available again in the memory pool, and acquisition can proceed. If this step is not done, or is faulty, the `rte_eth_rx_burst` function fails silently, and does not receive anything. The NIC's RX queues fill up, and when full, drop ingress traffic. This packet drop is one way to notice a faulty acquisition, and can be queried using the `rte_eth_stats_get` function.

1.1.3.2.3 Final cleanup This step mirrors the first step, and mainly consists in stopping the NIC and deallocating every allocated memory zone. Note that this step is a best practice and is strongly encouraged, but most modern OSes should be able to free allocated memory on their own when a process finishes its execution, and NIC drivers should be able to stop the NIC similarly.

1.1.4 GPU packet processing, DPDK

The second iteration of this acquisition system was designed with the goal to remove the CPU from the critical path as much as possible. Using recent features introduced by Nvidia specifically for the MLX5 driver, we were able to setup memory pools resident in GPU memory. By associating a GPU memory pool instead of a CPU memory pool to an RX queue, ingress traffic is seamlessly DMA'ed to a GPU. This is also called GPUDirect in the context of Nvidia hardware.

Note that two limitations apply to this approach: it has only been tested on Nvidia hardware, and requires a special Linux kernel module shipped by Nvidia, `nvidia-peermem`, in order to function correctly. This module would most likely need to be rewritten for this method to work on hardware from other vendors. This is presented in greater details in Section 3.1.

The only other modifications necessary, compared to the previous design, are steps 2 and 3 of the main loop. Indeed, code used to parse headers and use payloads is executed by a GPU in this design, and custom code has to be written for this purpose. Parallelizing efficiently these tasks can be a challenging tasks. Multiple solutions exist, and we mainly considered three options:

1. One thread per packet
2. One warp per packet
3. One block per warp

These different options provide a tradeoff between parallelization and latency. Mapping one thread per packet is the approach presented in the documentation of this DPDK feature. This is ideal for parallel header parsing, but we found that it was an issue for payload processing. Indeed, in the context of our data acquisition system, the most common task was to copy payloads to a

reserved, application-managed memory zone, and mapping one thread per packet does not permit coalesced memory accesses (*cf.* Section C.3), resulting in great performance loss. Moreover, using this option, packets must be received in a huge number at a time to use as many threads as possible.

The two remaining options are almost the same, both enabling coalesced memory accesses during the payload copy. The warp granularity makes it possible to process more packets at a time per block, while copying more slowly, compared to the block granularity. The warp granularity is also slightly more efficient, as header parsing is duplicated across a single warp (32 threads), compared to a whole block (variable size, commonly 256 or 512 threads). However, the block granularity should provide in theory a slightly lower latency per packet.

One issue is yet to be discussed: as presented in the previous section, the function used to acquire packets, `rte_eth_rx_burst`, returns instances of the `mbuf` structure. Even in the case of a GPUDirect setup, this structure is resident on the host, and as such cannot be accessed from the GPU without special care. Moreover, we did not succeed when trying to retro-engineer DPDK codebase in order to understand from which memory zone these `mbuf`s are allocated. Indeed, when allocating a GPU memory pool, the GPU memory zone is explicitly allocated, but an additional memory pool must be allocated implicitly to contain the `mbuf` structure itself, while the GPU memory pool only contains raw packets (corresponding to the `buf_addr` field of the `mbuf` structure).

Since the memory zone from which these `mbuf` structures are allocated is not known, it is not possible to pin it using `cudaHostRegister`, making it accessible from the GPU through the PCIe. It is not clear whether hugepages already behave as pinned memory, but in any case, we miss the information of where this memory zone is.

One workaround found was to manually allocate a pinned memory buffer, and to copy relevant information from `mbuf`s inside, before accessing from the packet processing GPU kernel.

Another relevant detail is the cost of launching a GPU kernel over and over. Indeed, the average kernel launch overhead is of the order of 10 µs. Because of this, launching the packet processing kernel for each received packet is out of question, and launching for a burst of packets is not ideal. Indeed, the overhead per packet is of $\frac{10\,\mu s}{\text{number of packets}}$, and at 100 Gbit/s, one packet is received each 0.72 µs. Because of this, a lower bound for the burst size at 100 Gbit/s is at 14 packets. Naturally, the actual computation time needed to process packets also has to be considered on top of this lower bound.

In order to remove completely this concern, we actually used a persistent kernel. Persistent kernels are a class of Compute Unified Device Architecture (CUDA) kernel that run a main loop by themselves, without being relaunched by the host. This comes with the advantage of providing minimum latency and no overhead, but requires more complex and handcrafted synchronization mechanisms, that make this approach less maintainable.

1.1.5 GPU packet processing, DPDK gpudev

Fortunately, during the course of this work, a new component was added in DPDK: the gpudev library. This library adds support for GPU devices in DPDK, however it is only supported on Nvidia hardware through the CUDA driver as of DPDK 23.07.

In particular, this library adds a *de facto* standard way to implement the persistent packet processing kernel introduced in the previous section. To do so, the library provides a structure called a communication list, that makes the transfer of relevant fields of `mbuf`s easier. Additionally, it provides flags used for manual CPU-GPU synchronization. All of these features made it possible to develop a maintainable, rather standard high-performance acquisition system.

Figure 1.2 gives a representation of this final acquisition system design, relying on a persistent packet processing kernel and GPUDirect, as a Unified Modelling Language (UML) activity diagram.

1.1.6 Packet processing persistent kernel design

The general steps of the packet processing kernel followed generally this progression:

- Poll communication list for a new burst of packets
- Iterate over packets of the communication list

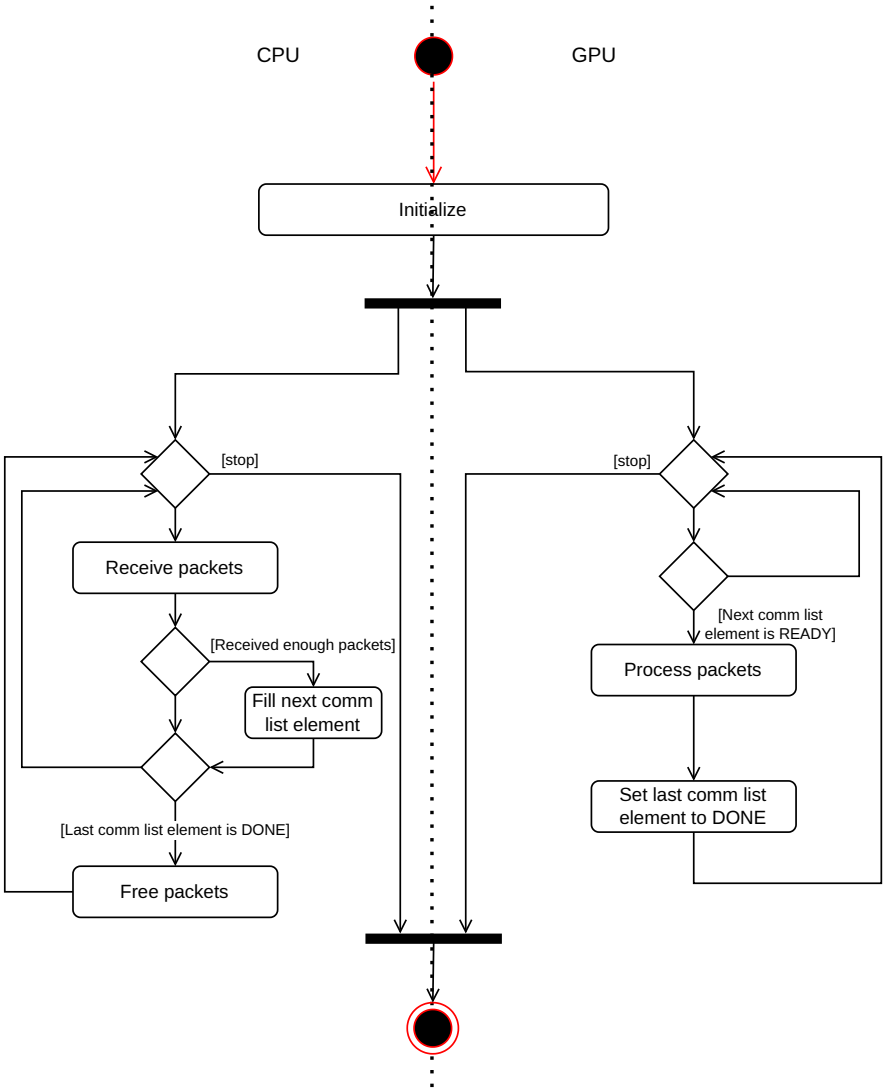


Figure 1.2: Acquisition system activity diagram. The communication list is a ring buffer, enabling CPU-GPU communication. Packet bursts are freed in an asynchronous fashion, unless the communication list is full, which can happen with a burst size too small.

  - Parse headers
  - Copy payload
- Notify through communication list that the burst has been processed

However, many important parameters of this kernel (complexity, duration, thread mapping) have to be tailored for each specific application. Because of this, the major part of this kernel has to be rewritten for each different application and protocol, reducing portability and increasing development time.

1.2 Performance tuning

Even with such architecture, and probably partly *because* it is elaborate and designed at low-level, special care must be taken before deploying on a real system. Various performance tuning tips are given in this section, making it possible to reach the maximum performance of the hardware.

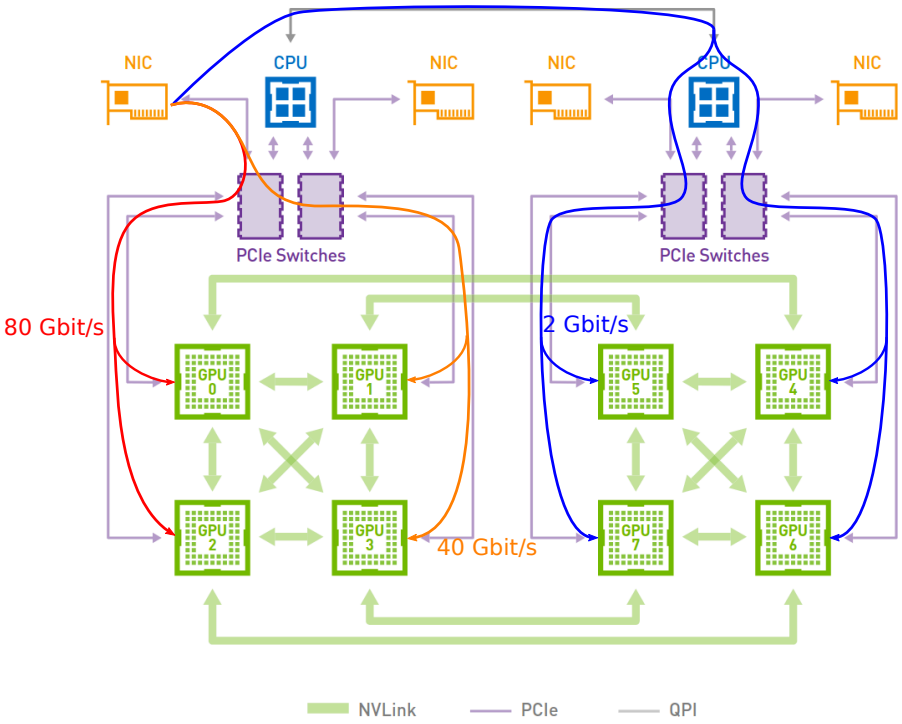


Figure 1.3: Maximum PCIe bandwidth on `moksha` along different routes. The shortest route features the best performance. `moksha` uses PCIe Generation (PCIe Gen) 3.0, making it more difficult to reach 100 Gbit/s.

1.2.1 PCIe topology

The PCIe network's performance is highly dependent on its topology[46]. In order to obtain maximum performance, two PCIe devices must be as close as possible in the topology. This was revealed in the first benchmarks we deployed, on a DGX-1 server, as shown on Figure 1.3. This specific experiment was made using a point to point configuration between two different servers, as represented on Figure 1.1b.

1.2.2 PCIe usage

Another factor that impacts the PCIe maximum bandwidth for a DMA is bus contention. Indeed, the PCIe bus is limited in terms of transfers per second (T/s). Consequently, small, repeated transactions, such as polling a flag can be strongly detrimental, and interfere strongly with critical transfers.

This has been observed on the first designs using the gpudev library, and GPU-side packet processing. A simple stop flag, used to control the execution of the persistent kernel described in Section 1.1.6, polled from the GPU but resident in host memory, reduced the maximum bandwidth from 100 Gbit/s to 4 Gbit/s. Note that for this first design, the GPU was polling as fast as possible, effectively flooding the PCIe under transfer requests.

Two solutions were devised to address issues of this kind. Figure 1.4 summarizes theses approaches, detailed more precisely in the next two subsections.

1.2.2.1 Reasonable transaction intensity

By adding sleeps of a few hundreds of microseconds in the busy waiting loop, it is possible to solve this problem almost entirely. A small overhead remains, inversely proportional to the sleep delay chosen, and forms a tradeoff between maximum bandwidth and maximum latency.

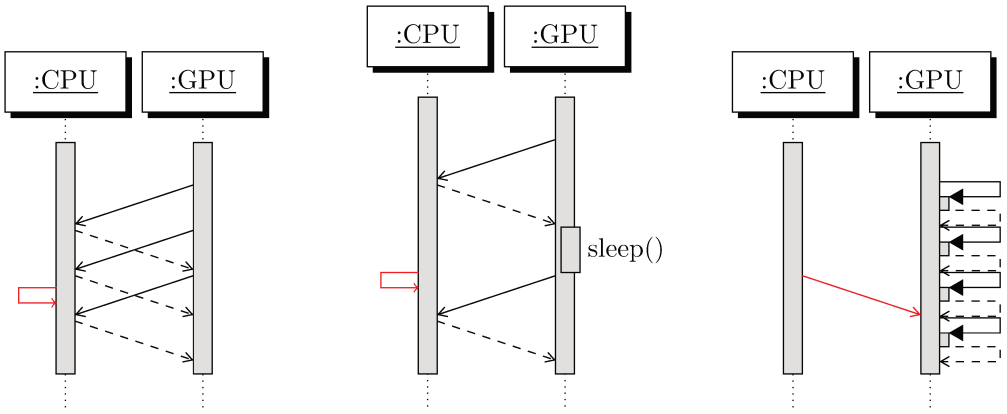
This approach is efficient and simple, but is not optimal, because of the aforementioned tradeoff.

1.2.2.2 Data locality

Another approach is to improve data locality. If a value, such as a flag, must be used multiple times from a GPU kernel, make it resident on the GPU. Conversely, if such use happens from the CPU, the corresponding data must be resident in host memory.

This approach makes it possible to reduce the number of PCIe transactions to their strict minimum, and to obtain the best performance possible on more critical parts, such as the NIC to GPU DMA of ingress traffic, containing the signal to be processed.

A combination of these two solutions can be used to reduce the load on the local memory system if necessary, and to reduce power consumption.



(a) CPU pinned host memory only (b) With sleep (c) Local data

Figure 1.4: Arrows between CPU and GPU correspond to a PCIe transaction. Red arrows depict setting the polled flag. On Panel (c), data locality is improved by polling a location in device memory, effectively reducing to a minimum the amount of CPU-GPU communication.

1.2.3 NUMA effects

Non-Uniform Memory Access (NUMA) materializes locality from the point of view of the host, and translates partially the constraints induced by the PCIe topology in a way that the Linux kernel can comprehend.

Paying attention to NUMA helps obtaining the best performance possible when communicating with the different devices of our system, such as NICs and GPUs. However, we did not notice significant performance improvements when carefully using the right NUMA nodes. This may be related to the fact that the CPU is removed from our data path, leading to potentially less negative effects.

1.2.4 Burst size

In the CPU main loop (left part of the activity diagram in Figure 1.2), the burst size is the number of packets received at once, using the `rte_eth_rx_burst` function of DPDK ("Receive packets" action).

This size plays an important role in the acquisition system's performance. A burst size too large increases latency, as some packets are enqueued for a longer time. In extreme cases, hardware RX queues of the NIC can even become full, leading to packet drops. However, a burst size too small results in more PCIe transactions, reducing the maximum bandwidth achievable.

A good value for the burst size can be obtained through the `default_rxportconf` field of the `rte_eth_dev_info` DPDK structure, already mentioned in paragraph 1.1.3.2.1. However, the `rte_eth_rx_burst` function will not wait for additional packets: it received `up to burst_size` packets, but can receive less. This behavior can become harmful, since in some cases this can cause much variability in the number of packets received, and as such increase jitter and energy consumption.

In order to address this problem, two solutions were explored:

- Adding a sleep instruction in the active wait for incoming packets
- Waiting for a certain minimum number of packets to be available before receiving

Using a combination of these solutions, CPU usage was reduced from 100 % (caused by active waiting), to less than 2 %, even on the most intense workloads (100 Gbit/s, no packet loss). However, impact on latency has not been measured, and the active waiting method has been kept when latency is a strong concern.

In the current version, the sleeping delay is arbitrarily set to 1 µs, and was found to provide good results. However, CPU usage could be even further reduced by performing a linear interpolation based on the instantaneous bandwidth. This would make it possible to give an estimate of the time needed to receive enough packets to make it worth processing, and thus an ideal sleep time.

1.3 Telemetry

We gather in the term "telemetry" all methods related to communications from and to a pipeline. When building increasingly complex real-time systems, the quality of the telemetry system is essential, in order to obtain observable and reliable systems.

Telemetry was used with two different goals for this work: data dumping and pipeline control and monitoring.

1.3.1 Data dumping

In order to provide observability for pipelines at different stages, we routed copies of streams of data out of the pipeline, and stored them to disk, as schematized on Figure 1.5.

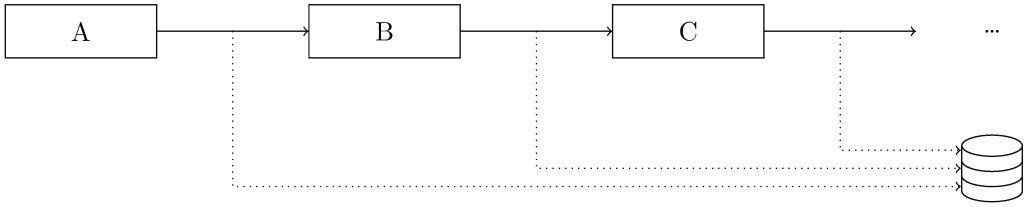


Figure 1.5: Data dumping in a mock pipeline

Three main aspects were desirable for this data dumping system:

- Low interference with pipeline, including:
  - Asynchronicity
  - Limited PCIe usage
- High bandwidth

Indeed, even if this data dumping system is critical to understand and debug what happens in the pipeline, it does not take a part in the computations or acquisition. Computations and acquisition must continue to run unchanged when this data dumping system is active. This comes with a constraint of asynchronicity (data transfers must not delay other operations of the pipeline) and limited PCIe usage (transfers must not bottleneck the PCIe bus and cause packet drops in the acquisition system). Finally, high bandwidth is required to keep up with real-time, considering the large volumes of data dumped. Even if data products are reduced compared to the raw input data stream, the bandwidth from pipeline to disk can still be considerable.

1.3.1.1 GPU → CPU → Disk

The first method used in order to implement this data dumping system relied on a bounce buffer in pinned host memory. Data was first copied using `cudaMemcpyAsync` from device to host, effectively recovering data asynchronously from the GPU. A second step was then required, a synchronous write from host memory to disk.

This method is asynchronous with respect to the GPU, but requires a synchronous call on the CPU, which is not a big issue. However, this double transfer adds unnecessary load on the PCIe, and was causing packet losses in the data acquisition system when dumping large amounts of data. This motivated a search for a GPU → disk DMA solution.

1.3.1.2 GPU → Disk (GPUDirect Storage)

Luckily, Nvidia once again simplifies this task by providing GPUDirect Storage (GDS), a library making it possible to setup GPU to Non-Volatile Memory Express (NVMe) disk DMA rather easily. We relied on this solution and removed the host bounce buffer, halving the load on the PCIe. No further packet losses were observed after switching to this system.

The setup of GDS follows a few simple steps:

- Allocate GPU memory
- Open a file
- Register both GPU memory and file descriptor for use with GDS

After these steps, a special call, analog to `cudaMemcpyAsync` (`cuFileWrite`), is used to write to a file from the device memory.

1.3.2 Pipeline control and monitoring

Some pipelines, especially in AO and radar, require live reconfiguration. Indeed, AO systems need to adapt to different observing conditions, such as a different wind speed or turbulence model. The most advanced radar systems, on their side, are heavily dynamic, changing of sampling frequency, mode of operation (vigilance, tracking, …) or countermeasure against jamming.

It is also very useful to obtain information about a running pipeline, regarding information such as current processed bandwidth, statistics of detections, *etc.*

Many solutions exist to provide this kind of feature, among which:

- File / Console Input/Output (I/O) (`stdin` / `stdout`)
- Bindings
- Network messages

In this work, a combination of these three methods have been used. Console output has been extensively used to provide debugging information and quick statistics about execution, as well as command line arguments for initial pipeline configuration. Files have been used for dumping / logging, as well as initial configuration through simple data storage formats such as JavaScript Object Notation (JSON). Python bindings have been used, especially in the context of COSMIC where they are abstracted and automatically generated. Finally, network messages have been used, especially for radar pipelines, where observability and control from an other machine was desirable.

In the rest of this section, we will briefly describe these different solutions.

1.3.2.1 File / console I/O

One of the most basic solutions to debug a program is to print its state at some specific point of the execution. In a more elaborate version, this type of inspection can form a logging system, essential to the understanding of a system, especially in case of bug or crash.

During this work, we considered a good practice to provide consistent logging. Fortunately, DPDK ships with a logging tool, that we used in order to take advantage of its interesting features, notably:

- Automatic formatting
- Message filtering
- Output location

This logging tool makes it possible to choose between different categories of messages, with a selection of default categories (malloc, mempool, power, …), and the possibility to create new categories. Different severities are available for messages too (debug, info, warn, error, …). The combination of these two message parameters is used both for automatic message formatting, and message filtering, leading to clearer and organized messages. Finally, it is possible to select the actual file used for printing these logging messages, being a special file such as `stdout` or `stderr`, respectively the standard output and error rendered on a console on regular OSes, or a text file that can be stored and accessed later on.

This type of approach is very simple to setup, however it is not very dynamic, can be difficult to integrate with other tooling such as a graphical interface, and is only well suited for the monitoring part.

1.3.2.2 Bindings

In the context of software development, binding is the act of making a codebase accessible from another programming language. This can be helpful to integrate quality legacy code into a newer language. A famous example of this is the collection of linear algebra routines initially written in Fortran, but accessible in most programming languages through bindings.

Another interesting use for bindings is to provide them for an interpreted language, such as Python or MATLAB. Interpreted languages, contrary to compiled languages (C++, CUDA, Fortran) are run by a specific program translating them to instructions on the fly, and not in a first separate stage. This makes them generally slower, but much easier to interact with, as they generally provide a console.

Binding CUDA compiled code to Python makes it possible to use extremely fast implementations in a console, with the possibility to change any parameter at request. However, in some sensible contexts, such as radar systems, this may be too permissive.

1.3.2.3 Network messages

When direct access to a machine is not possible, telemetry can be implemented as a networking protocol. Specific packets can be used to set parameters of the pipeline, and others can be sent by the pipeline to describe its current state.

This has the primary advantage of enabling stronger security, as the protocol is fully chosen and controlled by the implementer, and no full access to the machine is needed. The access to a machine can be controlled through this approach, this still gives an access point for a potential attacker, and the protocol must be designed with classical cybersecurity guidelines in mind. A second advantage is to fully uncouple the computation part from the data presentation part. Once the communication protocol is defined, both the pipeline and the user interface can be developed in parallel.

Regarding the implementation of this kind of telemetry methods, regular Linux sockets were used, as the bandwidths considered in this case are much smaller than that on which we used DPDK.

Now that the overall architecture of the data acquisition system has been presented, we can dive into how this architecture was adapted to the different applications we considered during this thesis.

Chapter 2

Applications

This data acquisition method was developed with a number of applications in mind: AO Real-Time Computer, FRB detection, and radar systems. Taking into account this diversity of applications, we aimed to design a generic and reusable data acquisition system, presented in the previous chapter.

However, there are significant differences between these different applications, ranging from rather simple ones, such as the variety of packet headers, to much more profound ones such as protocol structures. Indeed, even though all protocols considered aimed to transfer data in real-time, network protocols can support a wide range of features, namely multicasting, fragmentation, or reception acknowledgement.

In the context of this work, we identified two classes of protocols: streaming protocols, and bursted protocols. On the one side, streaming protocols send data continuously, and separate packets can be processed independently. On the other side, bursted protocols send bursts of packets, where a whole group of packets has to be received before completing packet processing. The second one was generally considered more difficult to process, as it requires complex synchronization behavior and/or a state machine.

In this chapter, we will start by presenting the common network protocols that were implemented, and then expand on the specifics of each data acquisition system, presenting what common parts were reused and what modifications were needed in order to suit the very diverse applications.

2.1 Common network protocols

Across all applications considered in this work, the first four layers of the Open Systems Interconnection model (OSI model) were common: Ethernet (L1 and L2), IP version 4 (IPv4) (L3), and UDP (L4). Table 2.1 reminds the different layers of OSI model.

Layer Number	Layer Name
7	Application layer
6	Presentation layer
5	Session layer
4	Transport layer
3	Network layer
2	Data link layer
1	Physical layer

Table 2.1: OSI model

In this section, we provide a description of these protocols, to use as a reference for what follows in the rest of the chapter.

2.1.1 Endianness

Most network protocols use the big-endian byte order. Care must be taken to respect this constraint, otherwise inconsistent results may arise.

The big-endian convention stores the Most Significant Byte (MSB) first, and Least Significant Byte (LSB) last in memory. On the contrary, the little-endian convention stores LSB first and MSB last. Table 2.2 gives an example integer stored using both in big-endian and little-endian conventions.

Byte index	0	1	2	3
Little-endian	`D2`	`02`	`96`	`49`
Big-endian	`49`	`96`	`02`	`D2`

Table 2.2: Big and little endian representations of the INT32 $1234567890_{10} = 499602D2_{16}$

2.1.2 Ethernet

The Ethernet protocol covers the first two layers of the OSI model: the physical layer and data link layer. Ethernet L1 is implemented directly in the NIC, and as such, we do not interact with it directly. Because of this, we will not dive into the details of the physical layer part of the Ethernet protocol.

The Ethernet L2 frame consists of a header presented in Table 2.3.

Field	dst MAC		(802.1Q tag)	EtherType	Payload	CRC
Size (B)	6	6	(4)	2	46–1500	4

Table 2.3: Ethernet frame[24]

Note that the 802.1Q tag, used to define Virtual LANs (VLANs), is optional and was not supported in our work. Other L2 protocols that could be substituted to Ethernet include:

- I²C: Mainly used with integrated circuits
- SpaceWire: Used in European Spatial Agency (ESA) spacecrafts
- USB: L2 protocol associated with the Universal Serial Bus
- PCIe: L2 protocol associated with the PCI Express
- …

These, among others, can be substituted to Ethernet in order to apply to different contexts, without interfering with higher OSI model layers.

2.1.3 IPv4

The IPv4 protocol is an L3 protocol, playing a core part on the internet, as it enables a logical addressing system and routing. It is a connection-less protocol, and does not provide any guarantee on the delivery of a packet. It supports unicast (one receiver), broadcast (all receivers), multicast (some receivers) and anycast (one receiver among a set of possible receivers).

The IPv4 frame follows the structure in Table 2.4. And the non-trivial fields of this header are described in the following list.

- IHL: Internet Header Length, the number of 32-bit words in the header (variable because of the Options field)
- ToS: Type of Service, used to request a specific performance focus: lowdelay, throughput, reliability or lowcost. Note that this field is being replaced by Differentiated Services Code Point (DSCP) and Explicit Congestion Notification (ECN).

Offsets	Byte	0								1								2								3							
Byte	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				TOS								Total length															
4	32	Identification																Flags			Fragment offset											
8	64	TTL								Protocol								Header checksum															
12	96	Source address																															
16	128	Destination address																															
20	160	(Options)																															
⋮	⋮																																
56	448																																

Table 2.4: IPv4 frame[25]

Offsets	Byte	0								1								2								3							
Byte	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Length																Checksum															

Table 2.5: UDP frame[60]

- Flags, Fragment offset } Used to handle fragmentation
- TTL: Time To Live, the remaining number of jumps allowed during routing
- Protocol: Upper (L4) protocol code[4]
- Options: Additional options[3] (not supported in this work)

The IPv4 protocol can also be substituted in favor of other L3 protocols, such as:

- IPv6: IP version 6, an upgraded version of IPv4, that has a greater address space and is progressively taking over IPv4 for consumer internet.
- IPsec: Internet Protocol Security, providing end-to-end security at L3
- …

2.1.4 UDP

Finally, UDP is used to send messages unreliably (no guarantee that a packet has been transferred) and without connection, at L4. It mainly adds a *port* field, that allows to identify the application being used (L7), and a checksum, used to verify data integrity. It is generally used for its very lightweight design making it very easy to implement, and predictable latency, due to its unreliability. Indeed, other protocols such as Transmission Control Protocol (TCP) provide a mechanism to resend messages that were not received, adding reliability at the cost of occasional slow-downs.

UDP is also unordered, thus packets may need to be reordered before processing.

Table 2.5 describes the header associated to this protocol.

Alternative L4 protocols include:

- TCP: Provides a reliable and ordered transfer over a connection between a client and a server, as well as congestion control and error checking, among other features
- UDP-Lite: Even less reliable than UDP, does not discard packets with a bad checksum, to let the user take action on bad packets
- SCTP: Stream Control Transmission Protocol, a newer protocol bringing together the best of TCP and UDP

- DCCP: Datagram Congestion Control Protocol, UDP with congestion control
- …

2.1.5 Note on alignment

The very common combination of Ethernet + IPv4 + UDP headers totals for 42 B. This particularity caused many problems for GPU packet processing, because of alignment. Indeed, by default using DPDK + GPUDirect, packets received are aligned to a 256 B[1]. However, with the additional offset of 42 B, the start of the UDP payload is aligned to only 2 B.

On CPU, reading or writing a value from a misaligned address is generally legal, even though it can in some rare case cause a misaligned address error, or more generally cause a performance degradation.

However, on Nvidia GPUs, misaligned address access is simply forbidden, and always results in an error. Care must be taken either to correctly read the misaligned address, or avoid them altogether. We will detail these two approaches in this section.

2.1.5.1 Multiple aligned I/O for one value

The simplest workaround, and direct approach, is to manually read (resp. write) multiple aligned addresses and perform bit arithmetic in order to reconstitute the value. A visual representation of a misaligned value and how to access it is given in Figure 2.1.

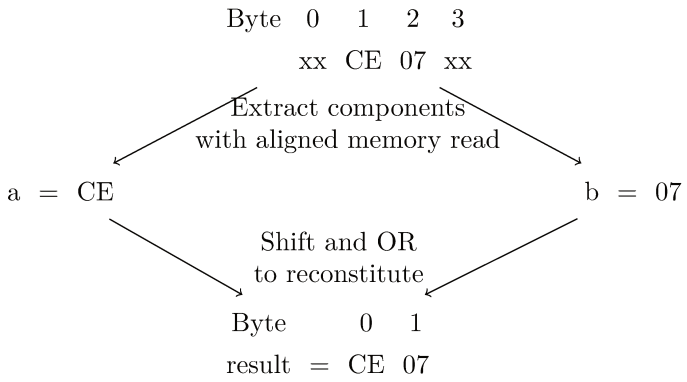


Figure 2.1: A misaligned 2-byte value, and a way to read it. Two bytes are read independently, respecting a smaller alignment, then recombined in registers. However, smaller memory accesses are less optimized on GPU. Consequently, access to misaligned values bears a strong overhead.

2.1.5.2 Multiple aligned I/O for multiple values

Another approach is to read multiple values surrounding the misaligned value to shared memory. This makes sense when working with networking headers, which are rather lightweight, and other values of the header will most likely have to be read (resp. wrote) at some point.

A perfect candidate for such task is `memcpy_async`, that takes care of the alignment automatically, using the highest alignment possible, and can bring other performance improvements.

In this approach, the header can be loaded (resp. stored) as is, or go through a chunked state in shared memory, in order to further enforce alignment..

[1] The default minimum alignment in CUDA for allocations is 256 B, and is kept by using memory pool elements of size multiple of 256 B

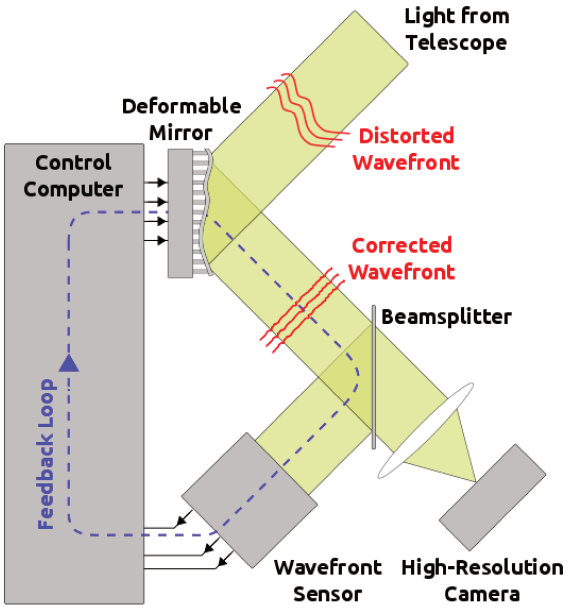
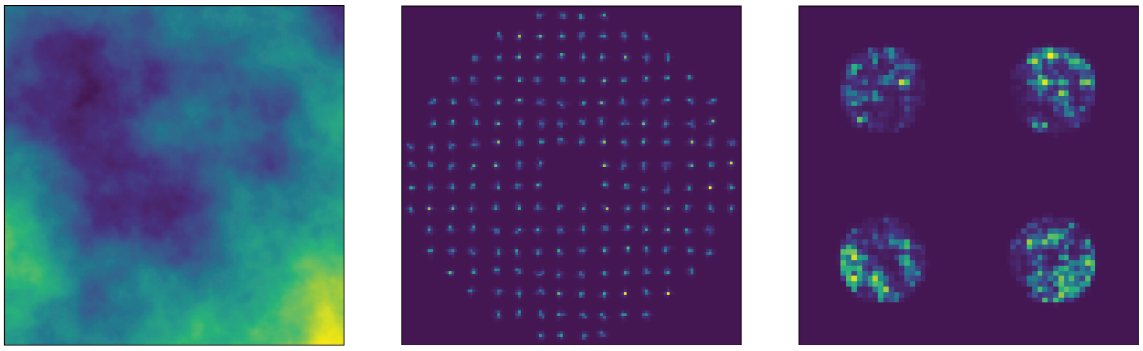


Figure 2.2: Typical AO control loop



(a) Simulated wavefront (b) Shack-Hartmann WFS (c) Pyramid WFS

Figure 2.3: Example response of two usual WFS (2.3a and 2.3b) to a specific wavefront (2.3c)

2.2 Adaptive Optics

AO is a technique used to increase the quality of images of a ground-based optical telescope. The crossing of the atmosphere disturbs incoming wavefronts, resulting in a blurred image. Because of the nature of the atmosphere, this blur is not uniform and is a function of time. In order to counter this perturbation, a control system is used: the AO system. Figure 2.2 gives an overview of a typical AO control loop.

The division of LESIA in which this work has been conducted (the HRA pole) is specialized in AO, and is notably involved and in some cases responsible for the development and maintenance of many AO RTC (denoted as "Control Computer" on Figure 2.2), such as the AO system at Keck, or the upcoming AO system Multi-AO Imaging Camera for Deep Observations (MICADO) of the Extremely Large Telescope (ELT). Latency is key here, as following up as closely as possible on the evolution of turbulence will maximize the AO performance, ensuring stable operations and optimized image quality.

In the context of this work, we focused on the acquisition to the RTC of images sent by the WFS. Figure 2.3 shows examples of typical images sent by WFSs.

In this section, we will expand on the different applications of our acquisition method in the context of AO.

Byte	0	1	2	3	4	5	6	7
0	topicId				componentId		applicationTag	
8	reserved		version		sampleId			
16	timestamp							
24	frameId		numFrames		payloadSize		reserved	

Table 2.6: MUDPI header

Byte	0	1	2	3
0	Frame Info	Extended Info Size	Protocol Version	
4	Status			

Frame Info

Bit	0	1	2	3	4	5	6	7
0	Packet Type			IsSimSource	Endianness	Reserved		

Status

Bit	0	1	2	3	4	5	6	7
0	Error Severity				Stream Error Status			
8	Error Code							
16	Reserved							
24								

Table 2.7: RTMS header

2.2.1 ESO network protocols

The European Southern Observatory (ESO), an organization developing and exploiting giant telescopes such as the Very Large Telescope (VLT) or ELT, designed custom network protocols to communicate between the different components of their instruments. In this section, we will describe these protocols, that had to be implemented in order to acquire data in the context of AO systems developed for ESO's instrumentation.

2.2.1.1 MUDPI

The first such protocol is called Multicast UDP Interface (MUDPI), and is used on top of the UDP protocol. The main goal of this protocol is to provide additional information about the position of a packet in a larger transfer. The protocol header is described in Table 2.6.

2.2.1.2 RTMS

The second ESO protocol used in the context of AO is the Real-Time MUDPI Stream (RTMS). It is built upon the MUDPI protocol, and aims to make real-time processing pipelines easier to build. The protocol header is described in Table 2.7.

Additionally, the protocol features a leader packet (Packet Type = 001_b) and a trailer packet (Packet Type = 100_b), designed to enable preparation from the receiver end. These special packets do not contain payload after the RTMS header. A frame transferred using this protocol consists in:

- 1 leader packet
- N payload packets, where N depends on the size of the frame
- 1 trailer packet

As such, RTMS is a bursted protocol, and information from every packet must be used in order to make sure a frame is correctly received (no missing packet, correct frame, …). Special care must

be taken with edge cases, such as on pipeline start, where the end of an incomplete frame may be received. In such an event, the frame is dropped.

2.2.1.3 Discussion

These two ESO protocols seem to be redundant with existing, widespread standards. In this section, we will discuss about these redundancies, and the perceived gains of using these protocols from our point of view.

Firstly, the main goal of these protocols is the real-time transfer of large (> Maximum Transmission Unit (MTU)) image frames. Consequently, image frames must be broken down across multiple packets on the emitter side, and reconstructed on the receiver side. This is very similar to Internet Protocol (IP) fragmentation.

Indeed, standard IPv4 fragmentation allows the transfer of up to 64 KiB, and IP version 6 (IPv6) fragmentation allows the transfer of up to 4 GiB. Moreover, because it is standard, it can benefit from standard support, from the OS or from other frameworks such as DPDK. Finally, this can be transparently used through UDP frames larger than the MTU, a type of jumbogram, automatically fragmented if using IPv4 or IPv6 as L3 protocol. This may be true for other L3 protocols as well.

The only missing features would be MUDPI's fields topicId, componentId, applicationTag, and timestamp, as well as RTMS's fields Status, IsSimSource and Endianness. All fields could compose a single application-specific header at the start of the jumbogram, except perhaps for MUDPI's timestamp, depending on the granularity desired. topicId, componentId and applicationTag could be encoded in the UDP's source or destination port, or different IP addresses.

The MUDPI protocol is enabling multicast, already enabled by the L3 IP protocol. Finally, real-time tuning is already possible with the IPv4 protocol, through the use of the Type of Service (ToS) flag.

A sample transmission implemented with jumbograms would save about 1 % of transmitted bytes compared to RTMS transfer. This is not significant, but worthwhile. However, the main advantage of such an approach would be the increased maintainability coming from a widespread standard, and potential hardware acceleration.

For all of these reasons, we failed to understand the motivation behind these additional protocol layers. But WFS are designed to emit their images using RTMS, so we need to support these protocols in any case.

2.2.2 ELT-MICADO

The first application of this work applies to MICADO, an upcoming AO system of the ELT.

MICADO will have three different WFSs[41], named ALICE, LISA and FREDA. The first two simply send a single image per cycle. FREDA consists of 3 different image types: Single Read (SR), Fowler Sampling (FS) and Double Correlated Sampling (DCS). Table 2.8 lists different characteristics of these WFSs. WFS image maximum frequency is of 2 kHz.

Name	Image dimensions	Data type	Payload Packets	Payload Size
ALICE	240×240	UINT16	60	960 B
LISA	800×800	UINT16	200	3200 B
FREDA SR	320×256	UINT16	64	2576 B
FREDA FS	320×256	UINT32	64	5136 B
FREDA DCS	$2 \times 320 \times 256$	UINT16	64	5136 B

Table 2.8: MICADO's WFSs

We currently support only ALICE and LISA, FREDA being slightly more complicated because of the three different image formats. Note that there is no real blocker to support FREDA, it is simply a matter of development time.

Two GPU packet processing kernels were designed for this application. The first one was based on packet splitting, acquiring the full packet headers on the host, and only the payload on

the device. The second one was performing the full packet processing on the device. These two approaches are described in this section.

2.2.2.1 Header parsing on CPU

Handling the RTMS protocol involves complex logic: three different kind of packets, possibly incomplete transmissions, and potential packet inversions. For simplicity, the first version of data acquisition for this application relied on CPU header processing, while still transmitting packet headers to the GPU.

The overall logic behind this packet processing CPU function is presented through an activity diagram in Figure 2.4.

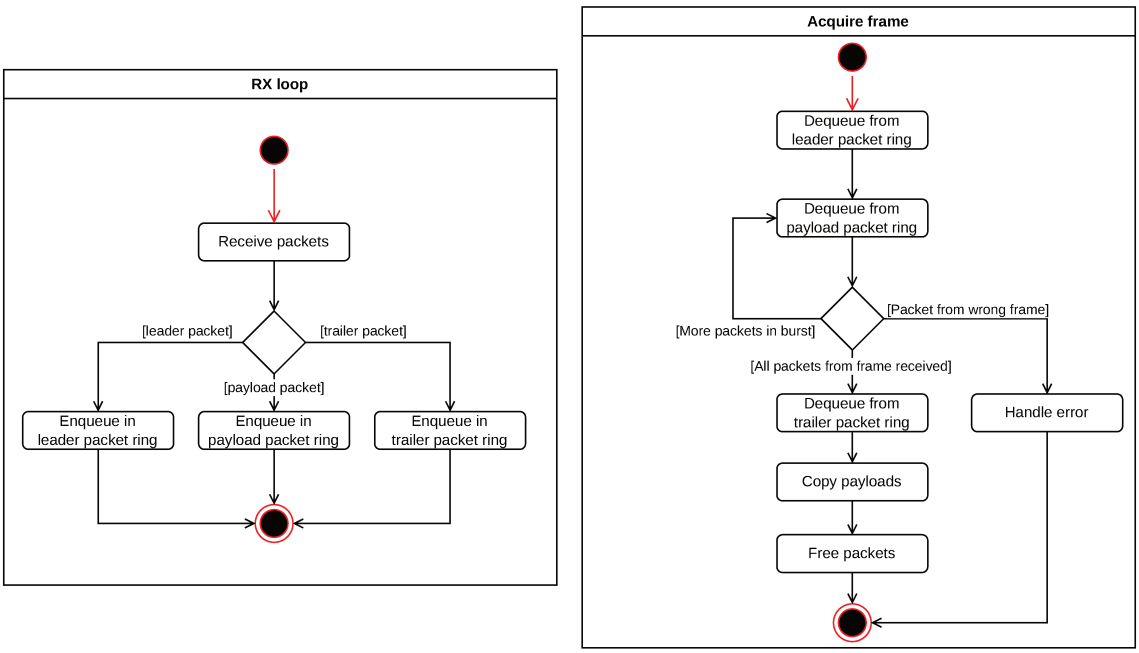


Figure 2.4: Activity diagram of the first AO packet processing function. Both groups run concurrently on different CPU cores.

Originally, triage between leader, payload and trailer packets was planned to be offloaded on the NIC, through the use of DPDK Flow API. However, even if DPDK provides support for user-defined protocol through RAW filtering items, the MLX5 driver does not implement it.

Because of this, we had to fall back on CPU-side triage, represented by the RX loop group on Figure 2.4. This loop acquires packets regardless of their type, and performs triage by enqueuing them in DPDK ring buffers, providing similar features to a First In, First Out (FIFO) queue.

2.2.2.2 Full packet parsing on GPU

The previous approach proved to be limited when scaling up to multiple WFSs, a use case motivated by the VLT-MAVIS instrument (see Section 2.2.3). Indeed, because of the very sequential nature of the first solution designed, parallelizing would require extensive threading, effectively dedicating one CPU core per WFS, delivering poor scalability, high energy usage and overall wasted memory and computing resources.

By replacing all sequential logic by states in a state machine, we were able to provide RTMS RX with packet processing directly on GPU, mapping one WFS to one CUDA block, thus taking advantage of the GPU's inherent parallelism.

In order to notify the host efficiently that a frame has finished reconstruction, we implemented packet processing with a semi-persistent kernel. This kernel is persistent for the time of a frame acquisition, and exits once the full frame has been received. Each iteration of the threads of the kernel advances both through the "Acquire frame" group of Figure 2.4 and different incoming

communication list elements, storing state from one step to another in a structure in global memory. This makes it possible to follow a rather sequential protocol while avoiding deadlocks and complicated branching, detrimental to GPU computing performance.

2.2.3 VLT-MAVIS

MCAO Assisted Visible Imager and Spectrograph (MAVIS) is an upgrade for the current AO system of the VLT. It will use Multi-Conjugate AO (MCAO), a technique combining the information from multiple WFS (8 in the case of MAVIS) to enable a wider field of view.

Indeed, with the classical AO approach, the correction is only valid in a very limited portion of the sky, around a few arcseconds. By using multiple WFS with different observing angles, it is possible to perform a tomographic reconstruction of the atmosphere, thus enabling AO correction in a much wider field of view.

We used the approach described in Section 2.2.2.2 in order to perform the acquisition of multiple WFSs.

2.2.4 Results

Being the first motivator for the development of this data acquisition system based on DPDK and GPUDirect, the AO use-case benefitted from the most thorough analysis.

The main focus in this case application being low-latency, we especially focused on latency, but also performed a quick bandwidth analysis.

2.2.4.1 Experimental configuration

Two configurations have been used in order to measure performance on this AO WFS frame acquisition system, the loopback (Figure 1.1a) and point to point (Figure 1.1b) configurations presented earlier.

In any case, a custom packet generator based on DPDK has been used. Since this specific part of the thesis, ESO released an official WFS simulator.

In the following paragraphs, we will expand on how metrics were measured for this pipeline, and how the two different configurations complemented each other.

2.2.4.1.1 Latency measurement Two types of latency measurement were set up during this work:

- End-to-end latency measurement: duration between the send of the leader packet of a frame and the end of packet reconstruction
- Last packet received to end of reconstruction

The first option is interesting to learn about the full time of transfer of an entire WFS frame. However, the transfer of a frame is often pipelined with the readout time of Charge-Coupled Device (CCD) image sensors, as represented on Figure 2.5. Because of this, the second latency measure mentioned is very interesting, as it is the only latency that will not be hidden by pipelining.

As mentioned in Section 1.1.1, measuring latencies between two machines is a difficult task. Indeed, a reference clock is generally missing. Two solutions apply:

- Clock synchronization. Two machines can be synchronized with enough accuracy to enable the measurement of latencies with sub-microsecond precision, through the help of protocols such as the Precision Time Protocol (PTP). However, we did not look into this solution because of limited hardware support, and existence of simpler solutions.
- Round-trip time measurement. By responding with an echo packet containing timestamps taken at each clock encountered on the data path, it is possible to deduce latencies, as shown graphically on Figure 2.6. However, this induces a strong modification of both the emitter and receiver components. Because latency benchmarking was not the main goal of this thesis, this was tried but support was dropped, as it added too much complexity. Moreover,

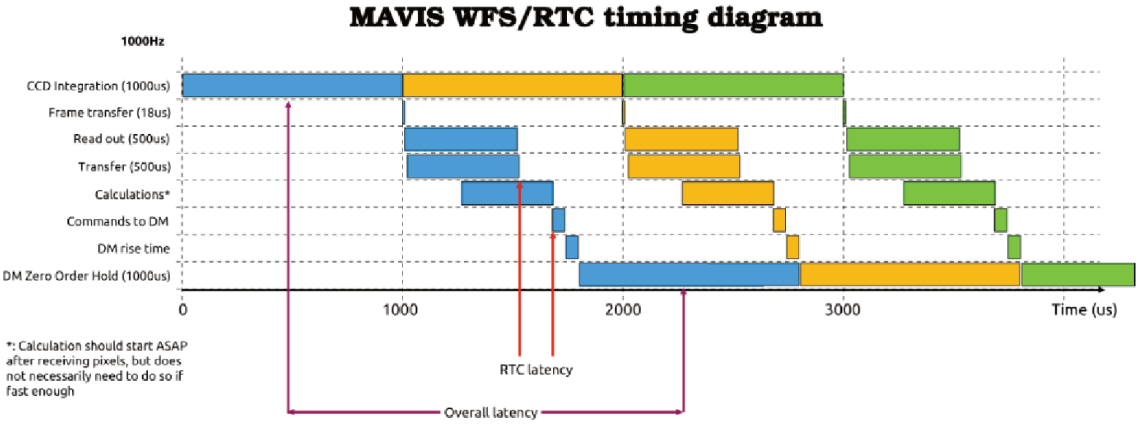


Figure 2.5: Pipelining on the MAVIS AO system[20] (courtesy of François Rigaut)

in the most extreme cases, such an approach may have created interference on the PCIe, and reduced apparent performance.

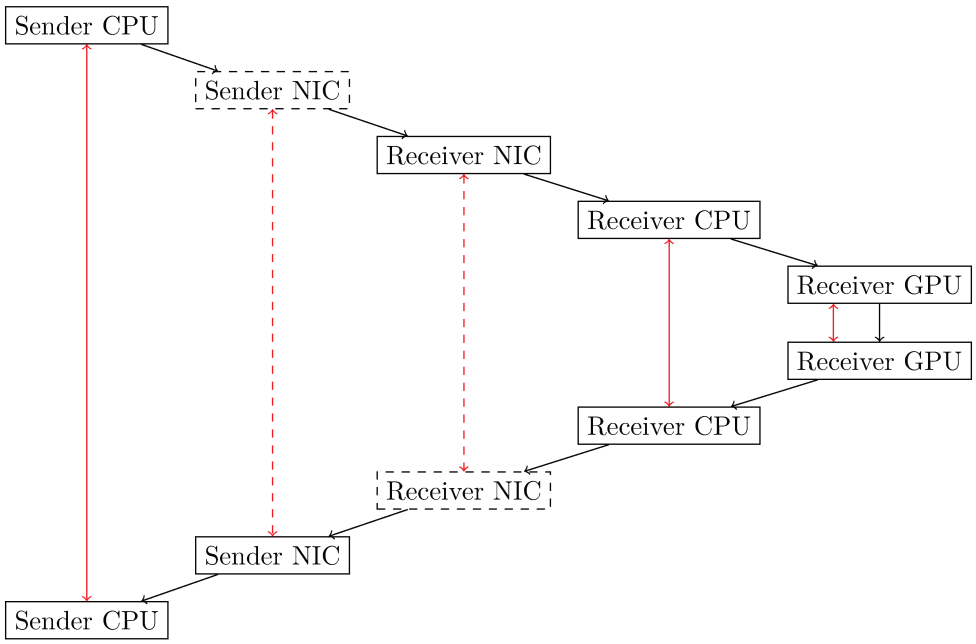


Figure 2.6: Latency measurement using round-trip time. Each cell represents a hardware component, equipped with its own clock. A timestamp is taken at each hardware unit. Time deltas, represented with the red arrows, can be calculated between two timestamps taken from the same hardware unit. Dashed lines indicate that a measure was not available, in this case because Nvidia NICs make it possible to take timestamps on RX but not on Transmission (TX) operations.

2.2.4.1.2 Bandwidth measurement The bandwidth measurements performed while testing this application were simply performed by dividing the payload bytes received over a certain period of time by this period of time, also called goodput in opposition to the whole throughput, which includes overheads of network protocol headers.

Bandwidth is much simpler to measure, as it can be inferred from a single point of observation, versus two for latency.

However, because of the experimental setup, in which whole frames are sent at once instead of pipelined, the achieved bandwidth is greater than the actual bandwidth that will be observed on

the deployed system. In this case, bandwidth is limited by the speed of the Ethernet link used.

2.2.4.2 Latency results

The first version of the acquisition system was tested on the MICADO case, with results shown on Figure 2.7 and Figure 2.8.

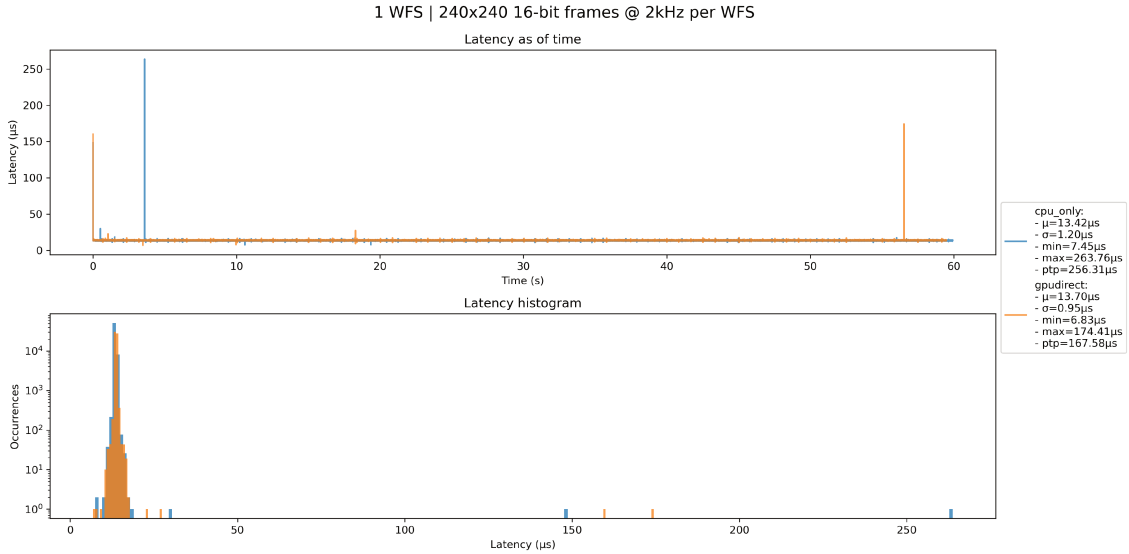


Figure 2.7: End-to-end latency on the MICADO case with first version of AO acquisition. Measured per packet, with the round-trip method.

The second version of the acquisition system was only benchmarked on the MAVIS case. Moreover, because of the design of this second version, it was difficult to measure the time between last packet received and end of packet processing, as packet processing happened on GPU. Instead, we measured the latency between the *send* of the last packet, and the end of packet processing. The results obtained this way are presented on Figure 2.9.

Notice the very low latency, comparable to that of the previous version, even with the overhead of the packet transmission taken into account. Note that we tried to estimate and subtract this overhead to give a fair comparison; however it is difficult to give a good estimate at this scale, considering the complex vectorization operations happening in TX and RX NICs. However, it is fair to say that this second version is at least as good as the previous one in regarding latency from last packet received to end of reconstruction, and is better in the sense that it supports acquisition from multiple WFSs.

These results are meeting expectations, as they represent only a few percents of the usual critical time available for AO RTCs, which is generally of the order of 200 µs.

As mentioned earlier, the ESO released an official WFS simulator, which plays a role of reference for testing. This was released at a late stage of this thesis, after this code was delivered to the MICADO team. However, they were able to confirm that this AO acquisition system was able to receive frames from ESO's WFS simulator successfully.

Because of these results, this method has already been selected for the MICADO RTC, with a first light planned for 2028, and is being considered for the same task on MAVIS.

2.3 Radio astronomy

Apart from AO, another domain in which this data acquisition system has been applied is radio astronomy. This domain also commonly requires the acquisition of large volumes of data, as it commonly relies on the exploitation of huge interferometers and/or very high time resolutions.

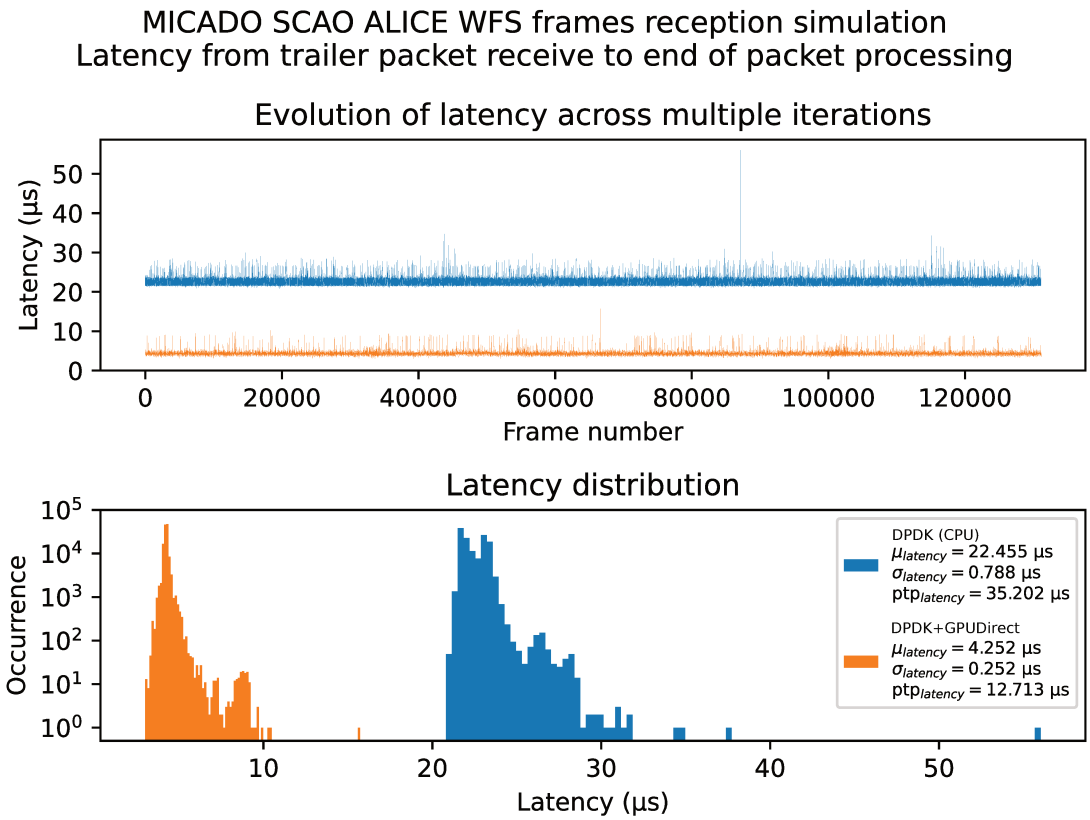


Figure 2.8: Latency from latest packet receive to end of reconstruction with first version of AO acquisition on the MICADO case

Two contributions were made in this domain. A side contribution was made as a collaboration with Commonwealth Scientific and Industrial Research Organisation (CSIRO) around the acquisition of large volumes of data in the context of an upgrade planned for Broadband Integrated Gpu Correlator for ATCA (BIGCAT), and the second concerning one of the core contributions of this thesis, the acquisition of NenuFAR Beamformed High Resolution (BHR) streams.

2.3.1 BIGCAT

An upgrade of BIGCAT, the correlator of Australian Telescope Compact Array (ATCA), is being developed in order to double the frequency band observed, increase flexibility on many aspects (spectral resolution, integration time, …), and provide better reliability.

In the current configuration planned, 16 nodes containing 2 GPUs each will need to acquire 60 Gbit/s in real-time. Current data acquisition prototypes are based on Linux's networking stack (see Section I), and rely on many fully used CPU cores in order to acquire such data stream, but is still unreliable.

CSIRO is exploring alternative solutions, such as FPGA-based acquisition (described in Section I), and we had the opportunity to discuss the possibility of assessing our DPDK-based approach.

A collaboration was started, and we managed to adapt our method to the application-specific protocol used on ATCA.

2.3.1.1 Results

We performed a simple port of the approach used on radars (Section 2.4), and transmitted this version to the team in charge of this development at CSIRO. On our simplified emitter, we were

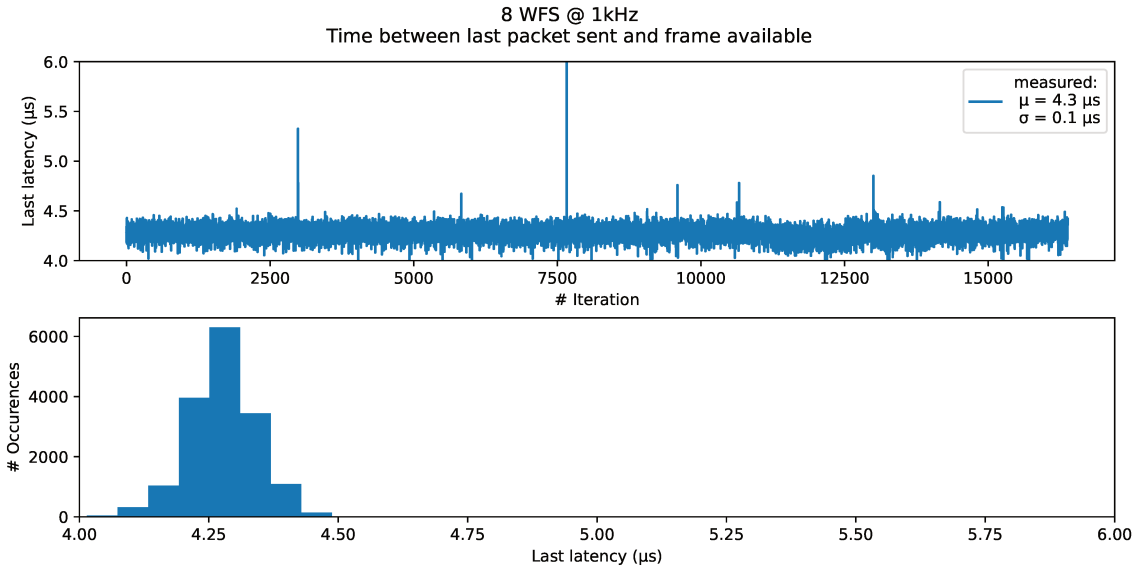


Figure 2.9: Latency from last packet sent to end of reconstruction with the second AO acquisition version, on the MAVIS case

able to get very close to 100 Gbit/s, with a loopback configuration (Figure 1.1a).

A more realistic emitter is being developed by this team in order to check the behavior of this prototype data acquisition system, but has not been finalized until now. Once this emitter will have been developed, our acquisition method may be seriously considered as a technical solution to data acquisition for this BIGCAT upgrade.

2.3.2 NenuFAR

One of the major applications of this thesis was dedicated to the telescope NenuFAR, and our data acquisition method was a central part of the FRB detection pipeline developed for NenuFAR.

In this section, we will describe the data transmission on NenuFAR, and explain how our data acquisition method was deployed.

2.3.2.1 Overview

The NenuFAR telescope is an interferometer, relying on beamforming in order to provide a usable signal. This beamforming stage is performed in an array of FPGAs called LaNewBa, and can be seen as a kind of edge computing: LaNewBa is located very close to the antennas, in a dedicated container. This beamformer can be configured to form up to 4 different beams, with tunable lower and higher observation frequencies and frequency resolution.

These beams are then multicasted over the network using IP multicast and UDP. Because the beamformer is able to output up to 10 Gbit/s, and that previous systems were based on the standard Linux networking stack, the signal is split over 4 different network streams called BHR, in order to make data acquisition possible. Note that these streams are purely related to the networking protocol at NenuFAR, called RSP_CEP, and does not map the 4 potential beams formed by LaNewBa.

An overview of the network surrounding NenuFAR is given in Figure 2.10. Thetis is the name of the server that we deployed on site, at Nançay.

2.3.2.2 Protocol

Each BHR stream can carry up to 192 frequency channels, for a maximum of 768 frequency channels taking into account the 4 BHR streams.

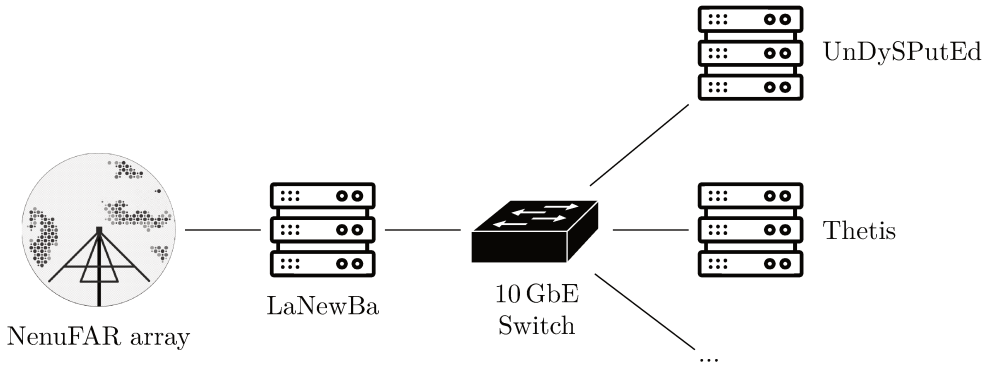


Figure 2.10: NenuFAR's network configuration overview

However, frequency channels from the actual beams produced by LaNewBa are more loosely constrained: only the total number of channels across all beams must be less or equal to 768. Frequency channels from beams fill that of the different BHR until full.

The BHR packets are UDP over multicasted IP, with a custom application-specific header described in Table 2.9.

Offsets	Byte	0								1								2								3							
Byte	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version Id								Source Info												# BpB				Configuration Id							
4	32	Station Id																# Beamlets per Bank								# Blocks							
8	64	Timestamp																															
12	96	Block Sequence Number																															

Table 2.9: RSP_CEP header

Note that NenuFAR is part of the broader project LOFAR, which explains the presence of some fields in the header that make less sense when using NenuFAR on its own. A breakdown of the different fields we considered is given in the following list:

- Number of Beamlets per Bank (BpB): Total number of frequency channels transmitted for each time sample. This is the sum of the number of beamlets per beam. This flag is broken down in two parts in the header, with the MSB coming first.
- Number of Blocks: Number of time samples in each packet
- Timestamp: Portable Operating System System Interface (POSIX) timestamp
- Block Sequence Number: Addition to Timestamp, providing sub-second precision. It is an integer value incremented each time a bank is produced, hence with a frequency of $\frac{200\,\mathrm{MHz}}{1024}$. However, since 200×10^6 is not divisible by 1024, this counter alternates between values of $\left\lfloor \frac{200e6}{1024} \right\rfloor$ and $\left\lceil \frac{200e6}{1024} \right\rceil$

Correspondence between beamlet index and actual frequency is given through a separate but complementary mechanism, linked to NenuFAR's planning. Indeed, because the beamforming operation is limited by computational power and Ethernet speed between LaNewBa and the receiving ends, a planning is set up to request the formation of specific beams, to observe specific events in the sky.

This configuration is stored as parset files, a format specific to NenuFAR, although very similar to INI files. An example parset file is given in Appendix B.

2.3.2.3 Acquisition system for NenuFAR

The acquisition system for NenuFAR is very simple, as the input is streamed. Dedicated ring buffers are filled on GPU, and once enough samples have been received, second-to-last chunks of

the ring buffer are processed.

This is a very simple and empirical way to account for potential packet inversion. However, no stronger mechanism has been implemented, as opposed to what has been done for radar data (Section 2.4.2.2), because the lower bandwidth of NenuFAR made it easy to allocate larger memory pools with more redundancy, and equally because the latency requirements were not as strong.

2.3.2.4 Results

The data acquisition system applied to NenuFAR successfully acquires packets without any packet loss over long period of time, even with data dumping as presented in Section 1.3.1. Data from NenuFAR can be acquired using a single 10 GbE port, and minimal CPU usage (<2 %).

This approach could be rather easily reused for other projects working around NenuFAR, as the acquisition system is self-contained in a `C++` class. However, this class forces the use of GPU computing and CUDA, which may not be desirable for some projects. Furthermore, the method of using ring buffers to store input data adds constraints on data access, not making this approach seamless nor plug-and-play.

The design process is only starting during this thesis towards a generic and portable way to deploy this data acquisition system. This is more extensively discussed in Section 3.3.

2.4 Radar

Finally, the last application of this data acquisition system was on radar data. Even though other applications were presented in chronological order, this one was developed continuously during the thesis, both benefiting from and inspirational to the developments of the acquisition system at a global scale, and is only developed now.

This data acquisition system was designed for an hypothetical radar, in order to comply with confidentiality aspects. Throughout this work, we considered a simple toy protocol, coined Simple Radar Protocol (SRP).

2.4.1 Protocol

The protocol was designed to be very lightweight and avoid the culprits encountered with ESO protocols, such as alignment issues, redundant fields and complex mechanisms. Packets can be processed independently from one another, and multiple streams can be sent at once, mapping either beams in the case of reception of already beamformed signal, or raw signal from radiating elements in the hypothetical case where beamforming would be performed as a part of the SP chain.

The protocol relies on the combination of Ethernet, IPv4 and UDP, as well as an application specific header described in Table 2.10. In order to enable best possible performance on GPU and simplest packet processing kernel, all fields of the SRP header are stored as little-endian values, as opposed to the usual network byte order. Because all Nvidia GPUs are little endian, this removes the overhead and added complexity of byte swapping. This is a significant design choice, but is not mandatory for GPU packet processing, as proved on every other application.

<table>
<tr><td>Offsets</td><td>Byte</td><td colspan="8">0</td><td colspan="8">1</td><td colspan="8">2</td><td colspan="8">3</td></tr>
<tr><td>Byte</td><td>Bit</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td>0</td><td>0</td><td colspan="16">Stream Id</td><td colspan="16">f_s</td></tr>
<tr><td>4</td><td>32</td><td colspan="16"></td><td colspan="16" rowspan="2">Timestamp</td></tr>
<tr><td>8</td><td>64</td><td colspan="16" rowspan="2"></td></tr>
<tr><td>12</td><td>96</td><td colspan="16">TSC Hz</td></tr>
<tr><td>16</td><td>128</td><td colspan="16" rowspan="2"></td></tr>
<tr><td>20</td><td>160</td></tr>
</table>

Table 2.10: SRP header

As can be seen, the protocol header has strange alignment of longer fields. This is to cope for the 42 B combination of Ethernet, IPv4 and UDP headers. Using the SRP header on top of this combination of underlying protocols, every field of the SRP header is aligned, and the following payload is aligned to 64 B, making vectorized memory access entirely possible.

In the SRP protocol, the index of a packet, used to reorder potentially inverted packets, is simply stored in the IPv4 Identification. Note that this is a repurposing of this field, normally used in the context of packet fragmentation. However, because this is a streaming protocol, no support for IP fragmentation is planned, and the flag can be reused.

UDP port numbers were originally used to encode the stream ID, but a specific field has been dedicated in the SRP header, since UDP ports may have been used for other partition tasks. Note that the total number of streams transmitted is not present in the header, and is assumed to be known beforehand, possibly with an additional communication protocol. This is to lift the constraint of supporting dynamic arrays, as this involves significant additional complexity in the allocation mechanism, both for receiving ring buffers but also further intermediate arrays.

The f_s field gives the frequency at which the values of the payload have been sampled. This is of great importance, as modern radars can modify their sampling frequency, in order to find the best compromise between electrical consumption ($\propto$ thermal dissipation) and detection performance (angular resolution, speed resolution, latency, …).

Finally, the Timestamp and TSC Hz provide time information, where Timestamp is a UINT64 counter, and TSC Hz the frequency in Hz at which the counter is implemented. This maps directly the time measurement APIs of DPDK, and the TSC Hz may be removed. However, this field aligns the payload to 64 B, *vs* 8 B without. As such, this field is important to performance, but may be replaced by another field if the frequency of the timestamp is fixed.

2.4.2 Acquisition system

The core of the acquisition system is very simple and does not diverge from what has been presented in Section 1.1.5. However, additions make it possible to provide additional security and performance to this method, as mentioned in the previous section. In this section, we will list these additions, and explain what motivated them.

2.4.2.1 Metadata ring buffer

In the context of this application, specifications imposed receiving bandwidths of up to 100 Gbit/s, with the goal of maintaining low latency. Because of this, the solution used on NenuFAR, relying on storing hundreds of milliseconds worth of signal before processing was not acceptable. Moreover, the use of less redundant ring buffer, coupled to a larger bandwidth, led to a risk of bad access to a ring buffer. A specific time sample could either not be already available, or already overwritten by the next pass on the ring buffer.

In order to improve security regarding this point, we provided an additional ring buffer, the metadata ring buffer, containing the full network header of ingress packets. At any point, this additional ring buffer can be queried to check the availability of specific time samples in the main data ring buffer.

This additional ring buffer is filled synchronously with regard to the data ring buffer, both filled by the packet processing persistent kernel presented in Section 1.1.6, adapted to the SRP protocol.

2.4.2.2 Safe data access

This metadata ring buffer provides a means for better security, although it is still tedious to check it at a regular pace. Moreover, as presented in the next part of this thesis, many stages related to radar Signal Processing could be implemented using CUDA toolkit libraries, where access to such exotic data structure is not supported.

In order to provide an easy interface to access the data ring buffer while checking the metadata ring buffer, we developed a safe access kernel.

This kernel once again copies data, from the data ring buffer to a chosen memory location, while checking the metadata ring buffer. The destination is meant to be a classic array, without any additional complexity, in order to make it usable with CUDA libraries.

The issue of this approach is that yet another overhead has to be paid in terms of kernel execution time and allocated memory. However, we found its impact to be non-negligible, but minimal in the different configurations considered.

2.4.3 Results

This application of our data acquisition system was mainly tested in point to point (Figure 1.1b) and switched loopback configurations (Figure 1.1c), with 100 GbE Nvidia Connectx-5 and Nvidia switch, Nvidia A40 GPUs, and PCIe Gen 4.0.

In the loopback configuration, we reached up to 99.7 Gbit/s of goodput over test campaigns lasting up to 8 h, without any packet loss, keeping less than 2 % CPU usage, and about 5 % of the GPU. This is a very encouraging result, as we managed to push our data acquisition up to line rate on 100 GbE, with limited resource consumption. This makes it possible to consider great scalability on faster Ethernet standards, although new bottlenecks may appear.

In the switched loopback configuration, we took advantage of the full duplex capabilities of the Ethernet links, and tested the sending and acquisition of 4×100 Gbit/s, effectively taking advantage of every piece of hardware available on this testing machine. However, this benchmark maxed out at 350 Gbit/s, and we were not able to identify precisely the origin of this limitation. The most plausible hypothesis is a bottleneck of the PCIe, as our data acquisition system requires PCIe transactions to function correctly, that could interfere with actual NIC to GPU DMAs at high bandwidth.

Chapter 3

Future work

3.1 Portability

As mentioned throughout this part of the thesis, all testing done in the context of this work was done on Nvidia hardware. Even though DPDK comes with a pledge of portability, it suffers from huge differences in the feature sets supported by each abstracted driver. This is especially true for the GPUDirect feature, that is explicitly supported only by Nvidia hardware and the MLX5 driver. Because of this, it would be interesting to define more clearly the boundaries of this acquisition system, on three different directions, listed in this section.

3.1.1 Other NIC + Nvidia GPU

It is not clear from documentation whether associating GPU-resident memory to a NIC using another driver than MLX5 is possible or not. As such, there is a non-zero probability that such configuration would work out-of-the-box. It might as well need support in the specific NIC driver. Such support could be developed by the vendor or by a user, and the complexity of such development is difficult to evaluate.

3.1.2 Nvidia NIC + other GPU

This other configuration should not work out-of-the box because of the probable lack of an equivalent to the `nvidia-peermem` module for other vendors. In this case, a port of the aforementioned module seems to be a good starting point to add support for other brands of GPU. Luckily, this module is open-source, which would make such port easier, if the feature set used in the module is roughly available with other GPU development frameworks, such as ROCm or Data Parallel C++ (DPC++).

3.1.3 Other NIC + other GPU

This option combines the two preceding ones, and can be considered as the most extreme. The suggestions of the two other ones can most likely be combined to make this option work. Indeed, thanks to the abstractions of DPDK, no further work should be required.

3.2 Alternatives to DPDK

In this work, we focused on the userland networking library DPDK. Taking advantage of the latest developments of MLX5, and its interface in DPDK, we were able to obtain great results, featuring a medium level of standardization and portability.

However, we could not remove completely interactions with the CPU, nor the need to use a small portion of the GPU's resources in order to perform the packet processing task.

In this section, we will provide an overview of the alternatives to DPDK that could be investigated to replace it, and cover their main pros and cons.

3.2.1 DPU

DOCA is a networking library by Nvidia. It is a DPDK rebrand with additional features, mainly centered around Data Processing Unit (DPU) control.

An interesting application of the DPU would be to use DOCA in order to perform packet acquisition and processing on a DPU directly. A DMA could then be triggered from the DPU to a GPU. In this design, the CPU could be completely removed from the critical path, and no additional work would have to be executed by the GPU.

However, it is not clear whether the DPU to GPU DMA is possible or not, nor how to configure it. We would also need a DPU with enough computing power to keep up with line rate. This is not guaranteed with the Nvidia Bluefield-2, but may become much easier with the upcoming converged DPU, featuring an integrated GPU.

3.2.2 GPUNetIO

Another cutting-edge component of DOCA is called GPUNetIO. It enables to trigger the NIC to GPU DMA from the GPU directly, without any interaction from the host. This makes it possible to remove completely CPU interaction in the acquisition process, and the associated CPU↔GPU communications, and comes with a pledge of simplifying system design, and increased performance and scalability.

However, the DOCA library is closed source and only usable with Nvidia hardware. Care must be taken regarding the dependency caused by building a project with such technology as a foundation.

3.2.3 Newer functionalities

In an effort to provide the most standard approach, the best solution is arguably to rely on functionalities of the Linux kernel. This enables the best integration with classical Linux administration tools, used to configure networking interface, routing, firewall, and so on.

As mentioned in Section I, the basic Linux networking stack is not enough to reach line rate on modern hardware. But new functionalities have been integrated in the Linux kernel in order to overcome the limitations of the previous API.

In this section, we will list the most promising such functionalities, and enumerate ideas on how to use these new APIs to replace the data acquisition method we presented.

3.2.3.1 Linux kernel module

The historical way to add functionalities to the Linux kernel is through the development of kernel modules. Linux kernel modules define new, custom system calls that can then be used from user space. This allows to develop new interactions with the Linux kernel.

Being executed in kernel space, a module has access to NIC driver control, as well as DMA programming. This may be combined to provide a specific `recv` function, transferring packets directly to GPU memory.

This could enable a vendor agnostic GPU packet acquisition. The only missing component would be a vendor agnostic GPU packet processing, but alternatives to CUDA listed in Section 8.1 could be considered.

3.2.3.2 eBPF / XDP

extended Berkeley Packet Filter (eBPF) is a solution to attach custom code execution to specific *hooks* in the Linux kernel. These hooks are specific execution points of the kernel, such as the start or the end of a system call. Being run in kernel space, eBPF programs can access very low-level APIs, such as the Linux DMAEngine.

The attached code must respect strong constraints to avoid major system instability. The two major constraints are that an eBPF program must not rely on looping, nor potentially crashing.

Because it interacts with critical components of the kernel, specific permissions are required to run an eBPF program. In the case of networking, the program must have two capabilities, `CAP_NET_ADMIN` and `CAP_BPF`. Please refer to Section C.2 to learn more about capabilities.

An eBPF program can interact with a userland program through a shared data structure called a *map*. We could use eBPF program to handle DMA from NIC to GPU on each `recv` system call, and as such replace the acquisition system we developed during this thesis.

eXpress Data Path (XDP) is a framework for eBPF, helping the development of packet processing applications. It makes it possible to use NICs drivers, as well as the other Linux kernel networking infrastructure, such as routing tables. However, this is not relevant if the packet processing task is run by a GPU.

Compared to modules, eBPF's instruction set is more limited, but its event-driven scheme makes it much easier to develop and integrate. Indeed, such an approach would seamlessly perform a GPU data acquisition for a specific network interface.

3.2.3.3 io_uring

io_uring enables handling Linux system calls asynchronously and through special rings shared between user and kernel space, in order to remove the overhead of context switching, while benefiting of the full Linux networking stack.

io_uring was initially designed for operation with storage, but has been extended to all I/Os including network operations. The preferred interface to io_uring is through the library `liburing`, abstracting some boilerplate code away.

This does not require specific capabilities and is designed to provide great performance, but relies fully on the Linux kernel, and as such does not rely on the GPU by default. A kernel module or eBPF program could be used in conjunction with io_uring in order to add the GPU packet processing capability, at the cost of increased complexity.

3.2.4 RDMA

A radically different approach would be to rely on Remote DMA (RDMA). This mechanism enables to perform memory operations over a network, given that both the specific protocol used is supported by both communication ends.

This is out of scope given our problem statement, which focuses on acquisition, without considering a modification of the emission mechanism. But this is still relevant, as it is a common way to transmit data to a GPU over the network.

RDMA transfers can typically be read requests or write requests over the network, the latter being possibly multicasted. Multiple questions arise regarding the use of RDMA in a data streaming application:

- Synchronization issues (could be solved using additional packets or busy waiting)
- Choice of the driving end (receiver side using read requests, or emitter side using write requests)
- Multicasting

These issues are not critical, but will have to be addressed if such an approach is considered for future work.

We will now proceed by describing briefly two major protocols enabling such transfer.

3.2.4.1 InfiniBand

InfiniBand (IB) is a full networking standard, replacing entirely L1 Ethernet. It is designed specifically for RDMA operations in HPC platforms. InfiniBand comes with its own set of protocols, but interoperability exists with usual protocols (Ethernet over InfiniBand (EoIB), Internet Protocol over InfiniBand (IPoIB)).

It requires specific support from the hardware, as the physical layer is different. Nvidia NICs support InfiniBand, through a specific configuration controlled with `mlxconfig` (parameter `LINK_TYPE_P1`).

It can provide transfers as fast as line rate. However, the main issue in using IB is the fact that it is not supported by any sensors manufacturer (either in optical or radio domains), would thus require a specific IP design to be embedded into the readout system of such sensor while being a closed source protocol, making it very unlikely that is will be supported in the future.

3.2.4.2 RoCE

RDMA over Converged Ethernet (RoCE) is another widespread RDMA protocol. It is an implementation of IP RDMA on top of a standard Ethernet network. It benefits from the wider support of Ethernet.

It is also able to provide transfers as fast as line rate[30], especially relying on hardware acceleration.

Similarly, `mlxconfig` can be used to toggle RoCE, through the parameter `ROCE_CONTROL`.

3.3 Encapsulation in a high-level component

Throughout the deployment of our data acquisition system over the multiple applications considered, we gained knowledge of the various constraints of such applications, and how they can become contradictory: bursted *vs* streamed protocols, focus on bandwidth *vs* latency, and so on. Because of this, even though we were able to reuse the same foundations, we failed to provide a clean, reusable data acquisition solution adaptable to any protocol. The two main blocking points for this task are:

- Variety of protocols
- Output data interface

The first point has already been addressed, and the second point relates to the fact that the ring buffers we used do not seem appropriate in the context of a reusable library, as their access is either unsafe because of the risk of accessing stale or overwritten data as is the case for NenuFAR, or because of the requirement of an additional copy kernel, as developed in the context of radar systems.

In order to handle most boilerplate code and make it possible to focus only on the protocol needing to be acquired, a solution could be to rely on the P4 language. P4 is a domain-specific programming language used to describe how to process packets in a generic way. It is designed to separate protocol specification from implementation. It can be seen as an analog of the Yet Another Compiler-Compiler (Yacc) language, transposed to the domain of packet processing.

The primary target of P4 has been programmable switches, but it is gaining capabilities for NIC control too, especially through Intel's `p4c-dpdk` P4 compiler targeting DPDK, bringing support to all NICs supporting DPDK, and multi-core CPUs execution.

Support for GPU-based application is still lacking, but an interesting goal would be to develop additional capabilities for the `p4c-dpdk` compiler, in order to enable GPU support. The same could be done for any alternative to DPDK as well, and would greatly help uncoupling application development from networking library choice, thus opening ways to more portability, flexibility and reusability.

Regarding the second point of the output data interface, one could consider options in the P4 compiler, to give choice in this tradeoff to the user. Other data presentation methods may be considered, such as event-driven programming, but have not been investigated in the context of this work.

Conclusion

In this part of the thesis, we presented a very high performance data acquisition solution based on DPDK and GPU packet processing, addressing many of the issues currently encountered while designing high-end cyberphysical systems, such as large astronomical telescopes and radar systems.

This data acquisition system matches our requirements in terms of reliability, high bandwidth and low latency, being able to receive a data rate close to line rate on 100 GbE hardware, featuring only a few microseconds of apparent latency (depending on the exact metric used), and capable of doing so during multiple hours without packet loss.

It vastly outperforms previous methods based on the standard Linux networking stack in terms of resource consumption, and went beyond our expectations in terms of energy consumption, requiring less than 2 % of a CPU core and about 5 % of a GPU to receive data at the maximum rate. This result is additionally very encouraging towards the scalability of this approach.

Finally, the final version of this data acquisition system relies on more standard hardware and software than previous FPGA-based approaches, while delivering comparable performance.

However, even though this system is based on COTS hardware, it still bears a strong dependency to Nvidia hardware, concerning both NICs and GPUs. The portability pledge of DPDK is non-trivial to put in practice, as it requires vendors to implement the interface proposed by DPDK. Core functionalities are almost always present, but more exotic ones such as NIC → GPU DMA are less likely to be supported. Because of this, a thorough analysis of alternatives to DPDK must be conducted in the future, although this task goes beyond the scope of this work.

Despite the limitations of the proposed data acquisition system, we were able to use it successfully in three domains and five specific applications: Adaptive Optics (MICADO, MAVIS), radioastronomy (BIGCAT, NenuFAR) and radar systems, leading to multiple achievements. Indeed, this system has been selected as the solution for data acquisition for the MICADO RTC, for which it has already been validated on the official network simulator, and with first light planned in 2028. It is also considered for the same task on MAVIS. Finally, it is already a part of a deployed pipeline for real-time FRB detection system on NenuFAR, described in greater details in the next part of this thesis.

Nevertheless, we stress the need of a fully converged solution across application domains, as many components of the data acquisition system have to be rewritten for each different protocol. This induces the development of error-prone boilerplate code, and longer development times. We raise the possibility of developing a new P4 compiler able to generate this boilerplate code following our data acquisition strategy, in order to make this method even more accessible.

Regardless of these considerations, once data has been received, only half of our goal is achieved. In the second part of this thesis, we will focus on the second half: GPU-accelerated computing.

Part II

High-performance GPU computing

Introduction

The topic of GPU computing is currently of great interest in the domain of HPC, and more generally in Information Technology. Indeed, since the emergence of programmable GPUs in the 2000s, GPUs have helped progress many fields of research through the rise of AI, but also by making their way into supercomputers with applications in scientific simulations, and embedded systems such as autonomous vehicles.

The current enthusiasm for GPU computing was perceived in the HRA pole of LESIA, and GPU-based astronomical instrumentation has been developed for a decade in order to provide real-time computing solutions for ever-growing AO systems. In the context of this work, our goal is to benefit from the high computing power density of GPUs in order to process a real-time stream of information at a high data rate, starting from 100 Gbit/s. Even if the memory bandwidth of GPUs is an order of magnitude larger than this, it can become very challenging to process these big volumes of data in real-time when applying a full and complex pipeline.

However, this domain in GPU computing is still a niche, which introduces some inconveniences. Firstly, tooling is much less developed than for the more regular CPUs. Even though a good compiler and debugger suite is available with CUDA (`nvcc`, `cuda-gdb`, `compute-sanitizer`), very good knowledge of GPU architecture is required in order to optimize GPU kernels efficiently, and deliver acceptable performance. Indeed, even though `nvcc` is very good at optimizing register usage and instructions, the CUDA language still requires developers to explicitly distribute work among different threads, and handle memory transfers manually between the different subunits listed in Section C.1. Secondly, libraries aside from the CUDA Toolkit generally lack maturity, and simple operations such as the 1D convolution, ubiquitous in this work, are hardly supported.

For the applications considered, a very high quality 1D convolution, and some other custom algorithms were required. Because of this, expert knowledge about GPU computing was acquired, and applied throughout the thesis, benefiting from regular exchanges and in-depth discussion with CUDA / DOCA architects. In this part of the thesis, we will dive into the optimization strategies that were applied in the context of this work, responding to a need of processing large volumes of data in real-time, while being as energy-efficient as possible.

In this part, after presenting the common methodologies that were applied throughout the rest of this work, we will especially focus on the two main domains of this thesis: radar systems and radioastronomy. In the context of radars, we first focused on the porting of an existing Signal Processing application for a Secondary Surveillance Radar radar, with the goal to understand the potential performance gains of a transition from CPU execution to GPU execution. In a second time, we focused on a generic Signal Processing application for primary radars, very demanding in terms of computational power. Finally, in the context of radioastronomy, we applied our GPU computing techniques to the detection of FRBs in real-time. Connections between radar systems in radioastronomy are discussed, as well as future research directions for this work.

Chapter 4

Methodology

Common methodologies were used in order to develop and study very high-performance GPU computing components. In this section, we expand on these methodologies, which will serve as a reference for the following chapters of this part of the thesis, regarding high-performance GPU computing.

4.1 CUDA kernel optimization

Throughout this thesis, we learned about many GPU kernel optimization techniques, through both literature and experience. In this section, we present the methodology developed and applied during this work in order to provide fast and efficient implementations.

Our method follows the Plan, Do, Act, Check (PDCA) methodology:

- Choose an algorithm / optimization
- Implement it
- Check results
- Measure performance

Optimizing a GPU kernel can bring significant performance improvement, but always up to a certain level, limited by an algorithm's complexity. Because of this carefully crafted algorithms are the first step of optimization. In the context of this work, most algorithms were found in literature, however some were modified for GPU execution.

The implementation part is rather straightforward, and is only a matter of writing valid and functional CUDA code.

Checking results is then a very important step, an implementation is of no use if it does not provide the good results. It can be tempting to measure performance before this step, but doing so may result in a loss of time if the results end up to be wrong. Unit testing is the best practice for this step, as it makes it possible to check results quickly once implemented, in a reproducible way, and can be reused at each iteration of the PDCA cycle.

Finally, measuring performance is a vast topic. Three main solutions exist at this point:

- CPU-based timer
- GPU-based timer (`cudaEvent`)
- Profiler

The two first options make it possible to measure one performance metric, the execution time, in a non-invasive way: these measures do not interfere with kernel execution. The use of GPU-based timers is generally considered more accurate, as it avoids common pitfalls and overheads.

The third option provides the most detailed information, especially with Nvidia Nsight Compute. However, this involves specific profiler-related measures during runtime, interfering with kernel execution.

4.2 Benchmarking with many degrees of freedom

One recurring problem in order to study and characterize our different implementations was the high number of free parameters involved. The number of dimensions for our benchmarking space was typically in the range 2–7. At the lower bound, 3-dimensional data (2 inputs and the output metric, the computation time most frequently) can already be challenging to represent, and may involve representations such as 3D surface or scatter plots, 2D contour plots or 2D heatmaps. These tools can be beneficial in order to give a visual idea of the relation between the different parameters, but fail to show precise trends in data, and are simply impossible to use on higher dimensional data.

Another issue is the exponential growth of the number of combinations of input parameters. Exploring the full parameter grid quickly becomes impossible with acceptable resolution.

4.2.1 Mathematical models

However, relations exist between these different parameters, and are in the case of this work rather simple. The complexity of the algorithms we implemented is most of the time known or easy to derive, and generally of the form $\mathcal{O}(n^{k_1} \log^{k_2} n)$, with $k_1, k_2 \in \mathbb{N}^2$. Because of this, it is possible, and relatively easy to find mathematical models describing the behavior of our implementations using linear models of the form $\alpha n^{k_1} \log^{k_2} n$. A simple comparison of the slope α can enable comparison of different implementations.

Nevertheless, care must be taken with the assumptions taken during grid exploration: specific GPU, data type, only powers of two, *etc.* Indeed, mathematical models provide an analytical continuation of experimental points, and may not capture small-scale, or non-covered effects, depending on the complexity of the model itself and of the parameter grid explored.

Finally, we were also able to significantly reduce the time taken for grid exploration: we achieved great precision using only a few tens of samples drew randomly from the parameter grid, as shown on Figure 4.1.

4.2.2 GPU saturation

In order for the samples to be representative, we need to saturate the GPU's Streaming Multiprocessors (SMs). Indeed, execution time plateaus until this saturation is reached and progresses in a linear fashion afterwards. An algorithm executing in this plateau zone is not desirable in any case, because it is a symptom of untapped GPU computing power, and not enough parallelism.

4.2.3 Uncertainty

Because these results are experimental, it is interesting to calculate the uncertainty that comes with it. In the case of a linear regression, this is not straightforward[9], and the confidence interval is given by:

$$\alpha \in \left[\widehat{\alpha} - s_{\widehat{\alpha}} t^*_{n-1}, \widehat{\alpha} + s_{\widehat{\alpha}} t^*_{n-1}\right] \tag{4.1}$$

Where $s_{\widehat{\alpha}} = \sqrt{\frac{1}{n-1} \frac{\sum_{i=1}^{n} (y_i - \widehat{\alpha} x_i)^2}{\sum_{i=1}^{n} (x_i - \bar{x})^2}}$ is the standard error of $\widehat{\alpha}$, and t^*_{n-1} is the $(1 - \gamma/2)$-th quantile of a Student t distribution with $n-1$ degrees of freedom, for a confidence level of $(1-\gamma)$. n is the number of points used for the linear regression.

We compute this interval in order to provide information about the reliability of the model, as an addition to the more commonly used coefficient of determination R^2. These metrics are also represented on Figure 4.1.

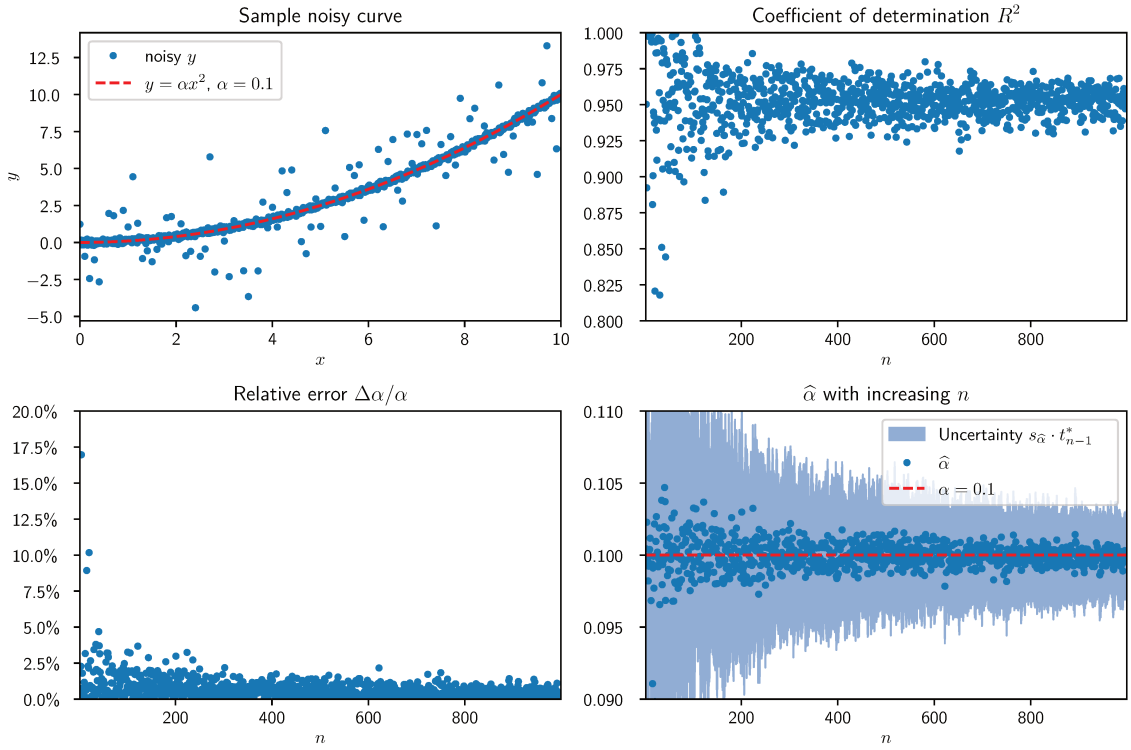


Figure 4.1: Sample curve and different metrics about linear regression

Chapter 5

Radar

In this chapter, we present the work that has been accomplished by applying high-performance computing techniques to radar systems. This work was two-fold. We started by porting an existing secondary radar SP on GPU, achieving limited improvements in terms of execution time; in a second time, drawing on the lessons learned from the first experiment, we implemented a primary radar SP from the ground up, enabling optimizations at a much bigger scale and with much more impact than in the first case.

You will learn about the limitations of a literal port from CPU to GPU, and about the motivation, or even the necessity of considering the bigger picture while striving for good performance while implementing an algorithm on GPU. Additionally, this chapter will describe the methodology adopted to optimize the various radar SP building blocks, starting from algorithm optimization to fine-grained optimizations based on profiling.

5.1 Porting an existing CPU-based radar SP

The first application considered in the domain of radar systems was that of porting the SP code of an existing, currently under exploitation, secondary radar system.

This application was chosen because it has existing, production-quality code, that could be used as a reference during development and performance study.

5.1.1 A secondary radar

A Secondary Surveillance Radar (SSR) is a special kind of radar, to be compared with primary and passive radars, other types developed at Thales and relevant in the air control management industry.

Secondary radars, additionally to being able to detect and measure distance of an object, are in charge of interacting with aircraft transponders, in order to request a number of information about them (identifier, altitude, speed, …). In comparison, primary and passive radars try to detect non-cooperative aircraft using their echo, respectively coming from the radar itself or from other radiation sources (the Sun, AM/FM radio, digital TV, cell towers, …).

Because of their interaction with transponders, secondary radars work in a very different way than others and in some ways adopt concepts and algorithms from the telecommunications industry. Notably, they need to support multiple protocols, that will be described in the next section.

5.1.1.1 Protocols

Multiple protocols have been created[22] in order to communicate between secondary radars and transponders, in order to query an aircraft about different parameters, or to perform surveillance for new aircraft entering the space covered by the radar. All radars share the same frequencies for these protocols: 1030 MHz for uplink, and 1090 MHz for downlink.

5.1.1.1.1 SSR The first and oldest such protocol is the Secondary Surveillance Radar (SSR). It is based on the emission of a series of pulses, separated by a variable number of microseconds.

In the case of uplink communication, from the secondary radar to the transponder, two pulses are used, P1 and P3. The duration between P1 and P3 maps to different requests, summarized in Table 5.1.

Mode	Duration between P1 and P3	Request
A	8.0(2) µs	Civil / Military Identification
B	17.0(2) µs	Similar to A
C	21.0(2) µs	Pressure altitude
D	25.0(2) µs	Not used
S	3.5(1) µs	Extended configuration

Table 5.1: SSR modes[22, pp. 2–1]

The downlink response to Mode A and C is based on the same kind of time modulation, encoding 4 octal values into series of pulses separated by 0.45(1) µs. Using this code, 4096 different values can be transmitted.

5.1.1.1.2 Mode S The Mode S (for Select) is a special extended interrogation mode, allowing for communication of other parameters, such as magnetic heading, airspeed, or vertical speed[23].

Additionally, Mode S makes it possible to interrogate multiple aircraft in a broadcast fashion (All-call) or single aircraft (Roll-call)[23].

5.1.1.1.3 ADS-B Finally, Automatic Dependent Surveillance-Broadcast (ADS-B) is an additional protocol introduced in the 2000s, consisting in aircraft broadcasting Global Navigation Satellite Systems (GNSS) information spontaneously and periodically, making it possible for air traffic controllers to locate and track aircraft precisely, without interrogation[22, sec. 5.2].

This position information are currently transmitted using the same protocol as Mode S and can be received by secondary radars, but could be transmitted differently in the future. Space-based ADS-B collection has been prototyped[13], which could help track aircraft in the most remote areas of the planet.

5.1.1.2 Interferences

These different communication protocols, as well as the volume of the air traffic nowadays (40.3 million flights per day before COVID), is the source of two different types of interferences in addition to classic RFI, described in this section.

5.1.1.2.1 FRUIT When a transponder replies to a request from a specific secondary radar, other secondary radars can receive the signal. These unwanted replies are called False Replies Unsynchronized with Interrogator Transmissions (FRUIT).

These interferences have to be identified and filtered out, in order to process correctly actual replies related to a secondary radar activity.

5.1.1.2.2 Garbling The same phenomenon can be coincidental with an actual request from a secondary radar: a wanted and unwanted response can be received at the same time, creating garble and potentially tamper with the actual transponder's response.

5.1.1.3 Secondary radar reception pattern

Most secondary radars, possesses multiple lobes of emission and reception, as illustrated on Figure 5.1.

In the case of the radar we considered, two directive lobes are pointing in front of the antenna with a slight offset, and two wider lobes, at the front and the back of the antenna's sensitivity, are used for control.

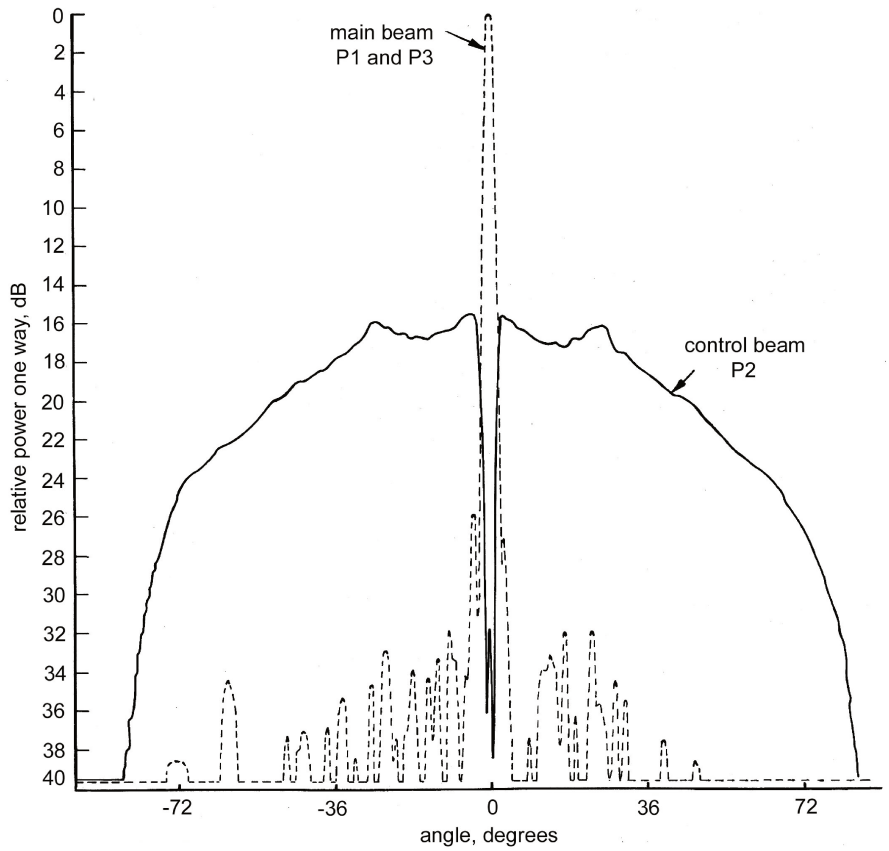


Figure 5.1: A secondary radar's reception pattern (By Monopulse01 M. C. Stevens - Own work, Public Domain, `https://commons.wikimedia.org/w/index.php?curid=12638575`)

These different lobes most importantly enable a precise measure of the direction of arrival, giving information on the azimuth of an aircraft, through ecartometry based on the two most directive lobes. Detection of interferences is done through the two control lobes.

Four complex signals, containing the two components In phase, Quadrature (IQ), are digitized from the antenna, and streamed to the Signal Processing component of the radar. The different signals are:

- Σ: Sum of the two directive lobes
- Δ: Difference of the two directive lobes
- Ω_f: Front control lobe
- Ω_b: Back control lobe

5.1.2 Existing SP implementation

Now that secondary radars have been presented, we can proceed by describing the existing SP implementation that has been ported to GPU.

5.1.2.1 History

The first generations of Secondary Radars were developed on custom hardware, programmed in FPGA or domain-specific Digital Signal Processors. They have evolved over three decades to use more and more COTS computing hardware, especially Intel PC CPUs whenever possible.

The latest generation of the radars run most of the Signal Processing tasks on multicore Intel CPUs. A GPU port is currently being investigated.

This legacy is the source of two issues:

- A sequential code, optimized for FPGA and suitable for CPU additionally taking advantage of the vector units available, but more difficult to optimize on GPU.
- The lack of vision about the whole SP. This work was developed starting from Functional Analysis Documents (FADs) and CPU code, which both describe the operations of the SP at a low-level, optimized for sequential logic.

Despite these issues, we managed to provide a working SP implementation, without prior CUDA knowledge, which is encouraging as regards to the average learning curve of CUDA.

Now that the history of this SP has been briefly described, we can expand on its different components. Two main components can be isolated: on the one side, the Video RX (VRX) is in charge of signal reception and preprocessing, that is done for all input signals, identical for all protocols presented above; on the other side, the Video Processing Pipeline (VPP) is different for different protocols, and multiple VPPs can be launched in parallel in order to process responses from different aircraft concurrently.

5.1.2.2 VRX

The VRX is the first stage of the SP. It is in charge of:

- Reading a signal frame in a special ring buffer managed by a data acquisition system, a frontend to what we presented in the first part of this thesis
- Convert input signal from FP16 to FP32
- Perform a strided convolution, in order to filter and decimate the input signal
- Take the Logarithm of the Modulus (LogMod) of the convolution, the decibel representation of the energy of the signal: $f(x) = 20\log_{10}(|z|) - 90$.
- Quantize the result to fixed point value, represented as an INT16

The conversion from FP16 to FP32 is required in order to have enough precision available during the convolution. This convolution is strided with a stride of 2 in order to halve the input data rate. This was also a constraint of the previous versions, that could be lifted by using GPU computing.

The LogMod is used here to reduce the dynamic range of the data, in order to be able to make it fit into an integer. The integer format is required later on, as the SP is designed with bit logic as an optimization.

After the VRX, the VPP processes incoming signal in order to extract a potential transponder response.

5.1.2.3 VPP

The VPP processes transponder responses, following an interrogation from the radar itself. Because of this, the radar expects a specific signal depending on what it just transmitted, which removes the complicated task of signal classification.

However, as explained in Section 5.1.1.2, great care must be taken with interferences, and much effort in the different VPPs is spent on extracting pulses out of background noise, and ungarbling. This involves the manipulation of discrete elements such as leading, falling edges and plateaus, which can be described using boolean values (presence / absence).

5.1.3 Implementation

In this section, we will expand on the main blocks implemented in the context of this application. Because of the many different steps, and the sensitive nature of this work, we will focus on abstracted operations.

5.1.3.1 RLE

Run-Length Encoding (RLE) is essentially a lossless compression method, efficient on signals featuring plateaus. An example of this algorithm is given in Table 5.2.

Input	0	0	0	1	1	-5	-5	-5	-5	-5

Value	0	1	-5
Length	3	2	5

Table 5.2: RLE applied to a short example sequence. Notice how the number of elements needed to store the input is reduced from 10 to 6 elements.

In the context of secondary radar SP, this is mainly used to find plateaus in boolean data, *i.e.* sustained state on specific conditions. This is extensively to identify time deltas in the input signal, and making it possible to parse the response from transponders.

The RLE encoding algorithm is sequential by nature, making it challenging to implement efficiently on GPU, requiring parallel algorithms to perform correctly. However, many sequential algorithms can be parallelized based on the prefix sum operation; this was done successfully for RLE[6].

In our implementation, we took advantage of the boolean nature of our input data to further optimize the parallel RLE algorithm. Indeed, by using boolean data, we can get rid almost completely of the array of values, as only the starting value (0 or 1) is required. However, the prefix sums of the algorithm still need to be performed.

5.1.3.2 Bitsets

While using boolean values can enable some optimizations as presented in the previous section, they can become challenging to store. Indeed, most modern computers store data as bytes, while boolean values can be encoded using a single bit.

Two solutions are available to store boolean data:

- Storing a single boolean value in a bigger data type: it is possible to store only two values of the 256 available in a byte, thus providing a simple solution to boolean value storage, at the cost of wasted memory.
- Storing multiple boolean values in a bigger data type: by relying on bit arithmetic, it is possible to store boolean values in a vectorized fashion, concatenated in a bigger type. This is commonly called a bitset.

Because of the ubiquity of boolean logic in the ported SP, most probably inherited from its FPGA roots, we based many implementations on a bitset data type, able to store and access boolean values efficiently.

Fortunately, this made it possible to implement very efficiently some operators, that could be applied directly on the bigger data type (AND, OR, XOR). However, some operations such as bit shifts were more difficult to implement, because of the need to transfer edge bits from one container type to another. This specific operation made some implementations massively more complex, and was a strong limiting factor for performance.

5.1.4 Results

The overall speedup for end-to-end processing was about a factor of 2, as reported in Table 5.3. This is much less than expected, as speedups when porting signal processing applications are commonly of two orders of magnitude[59, 11, 51]. This is due to the multiple limitations already mentioned about the ubiquity of bit arithmetic and sequential algorithms, inherited from previous implementations.

As can be seen on the execution timeline of Figure 5.2, the main bottleneck is the pulse detection algorithm, which takes care of cleaning as much as possible the input signal from interferences,

Mode	CPU (Intel® Core™ i7-6820HQ @ 2.70GHz)	GPU (Quadro M3000M)
SSR	22 ms	11 ms

Table 5.3: Full execution time comparison between reference (CPU) and proposed pipeline (GPU) for each mode

and relies most heavily on bitsets and conditionals. It can also be noticed that significant time is taken for RLE, suggesting that this algorithm could benefit from more optimizations, if possible. However, other blocks do not take significant time to complete. A heterogeneous solution could be considered, where RLE encoding and pulse detection are performed on CPU, and other operations on GPU.

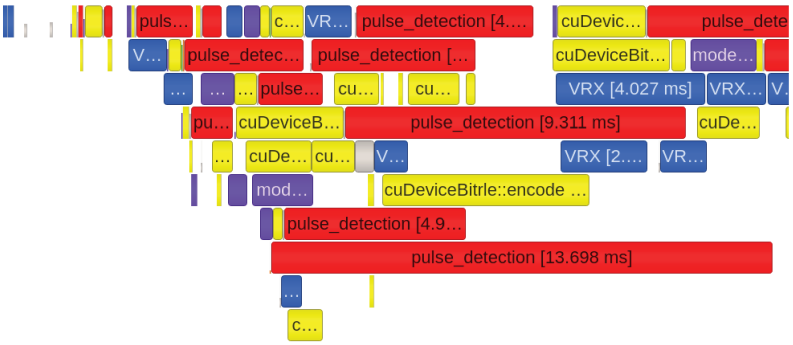


Figure 5.2: Timeline of execution of our CUDA SP, with time going from left to right. The vertical axis contains the different functions executing or waiting for execution at a given time, revealing pipelining. A first wave of functions is launched to process one SSR response, then a next wave comes for another, longer response. Note how computations can be pipelined to increase GPU usage. However, this causes contention on the GPU, and increases latency at the same time.

5.1.5 Future work / Lessons learned

GPUs may have the potential to revolutionize secondary radar SPs, and have already proven capable of halving the computation time of the whole pipeline, and even more for many components, such as in the case of the VRX.

The real bottleneck is the pulse detection algorithm, that heavily relies on bit logic and conditionals. However, a similar behavior could be implemented using a filterbank and as such convolutions, which are much better suited for GPU computing.

This relates to the famous quote from Donald Knuth:

> Premature optimization is the root of all evil ([32])

Because the specification of the VPP is already optimized for an FPGA target and for Single Instruction, Multiple Data (SIMD) (see Section 5.1.2.1), and that we focused on porting the exact behavior of the CPU version, we were not able to obtain best performance from the GPU.

A better implementation of parallel RLE, and a new algorithm for Mode S pulse detection adapted to GPU architecture are two logical improvements for this work, and may help to make GPU computing beneficial for Secondary Surveillance Radar.

A possibility for a more adapted pulse detection on GPU may be the use of a filterbank, if enough to handle all possible interferences. This would improve maintainability, as efforts could be focused on a single operation: the convolution. This filterbank approach has been extensively developed in the next part of this work related to primary radar systems, especially in Section 5.2.3.1, presenting different iterations of a high-performance convolution implementation on GPU.

5.2 Primary radar

A second part of this work was to focus on primary radar SP, which features higher computational intensity and less logical complexity, making it theoretically better suited for GPU computing. The scope of this work covers Pulsed Doppler Radars.

Drawing on our experience on secondary radar, and the mixed results we obtained in term of computation time improvement porting from CPU to GPU, we adopted a different approach for primary radar. Instead of porting existing optimized CPU code, we started back from the main algorithms.

For context, an overview of a complete primary radar pipeline is given in Figure 5.3.

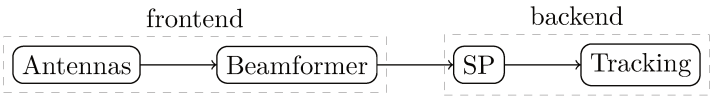


Figure 5.3: Primary radar full processing chain overview

Analog-to-Digital Converters (ADCs) behind each radiating element of the antenna sample the electromagnetic field with a given sampling frequency f_s, which vary from one radar to another, and can even be tuned in real-time on the most advanced radars. In current radars, this signal is beamformed on FPGAs near the antennas, before being sent over an Ethernet network with a UDP-like protocol. This part is called *frontend*, as it is the nearest from the antennas.

In contrast, the *backend* is composed of the Signal Processing stage, with the goal of detecting potential targets, and tracking, that aggregates these potential targets, in order to predict trajectories, identify maneuvers, and so on.

Note that the line between frontend and backend is rather blurry; current radars are built this way, but one perspective opened by our work is to move the beamforming to the backend, in order to make it even more adaptive, thus changing the delimitation between frontend and backend.

5.2.1 Classical primary radar SP

Existing radar SP at Thales are based on multiple CPU nodes. The number of hypotheses is limited by the computing power of the CPU nodes, even with optimized versions. The original implementation is manually optimized using SIMD instructions (Advanced Vector eXtensions (AVX) 2 or Streaming SIMD Extensions (SSE)).

Figure 5.4 gives an overview of the typical SP used at Thales for primary radars, and the rest of this section expands on the different building blocks listed in this figure.

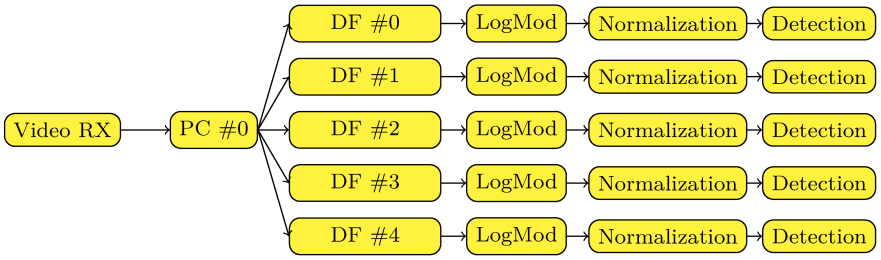


Figure 5.4: Classical primary radar SP

Each branch can be called an *hypothesis*, as different filters from a filterbank are used, matching specific parameters each time such as pulse shape, speed, acceleration and so on. An hypothesis about the actual value of these parameters is done. This can also be seen as a feature extraction technique.

5.2.1.1 Video RX

In the context of primary radar, VRX simply denotes the acquisition stage, and on optional type cast depending on the specific quantization used by the beamformer. Indeed, in order to save

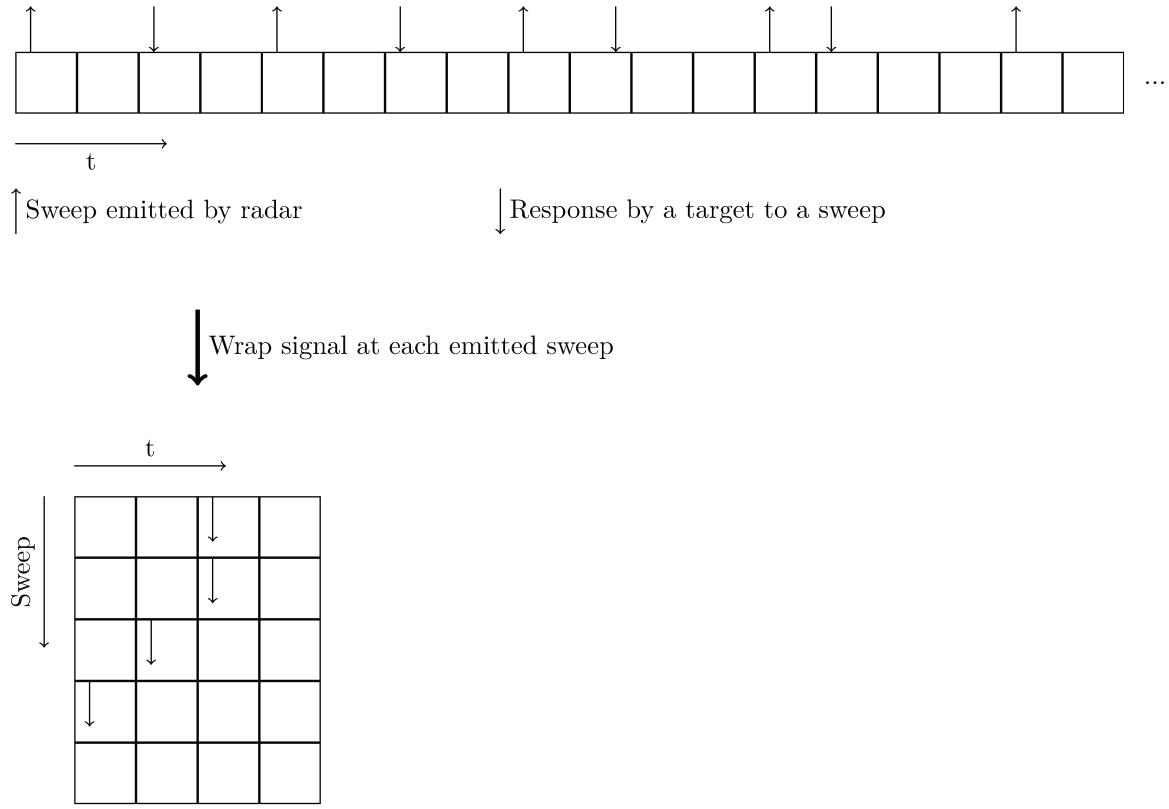


Figure 5.5: Depiction of radar SP input data. Notice how it is folded periodically at each emitted burst, making it possible to visualize more easily the migration from a target between different cells.

bandwidth, beamformed data can be stored in multiple formats, including:

- Fixed point samples stored in `INT<8/16/32/64>`
- Floating point samples stored in `FP<16/32/64>`

A conversion to a common data type such as `FP32` may be needed, in order to simplify downstream implementations and to have a better control about overflow and rounding errors.

5.2.1.2 Pulse Compression

In radar, Pulse Compression is the process of matching the previously emitted pulse shape in the received signal[31]. Indeed, most modern primary radars emit relatively long and faint pulses, compared to the first radars emitting very intense and short pulses. Moreover, the pulse follows a given pattern, in order to reduce the radar's sensitivity to interferences caused by jamming, other radars, or natural phenomenons. The Pulse Compression consists in a matched filter[57]: the received signal is convolved with the expected pulse shape. This filter responds significantly only on the expected pulse pattern. It is in essence very similar to Doppler Filters, although more simple as they are only one-dimensional. An illustration of matched filtering in the context of Doppler Filtering is given in Figure 5.6.

5.2.1.3 Doppler Filtering

Doppler filtering is the process of applying a filterbank to the result of PC, over the *repetition axis* (see Figure 5.5). Each filter is matched for a specific speed and acceleration.

In its simplest form, it is a matrix multiplication between a matrix of filters and the pulse compressed signal. However, current efforts to increase resolution increase the probability of range cell migration, making this simple form obsolete.

In order to support range cell migration, Doppler Filtering can expanded into a convolution, with coefficients of the filters spread onto multiple range cells.

The Doppler Filtering stage computes the following equation:

$$DF_{i_{HDF},i_{DF},i_{RC}} = \sum_{i_{Sweep}=0}^{N_{Sweep}-1} \sum_{j_{RC}=-\lfloor \frac{S}{2} \rfloor}^{\lfloor \frac{S}{2} \rfloor} IQ_{i_{Sweep},(i_{RC}+j_{RC})} \cdot h_{i_{HDF},i_{DF},i_{Sweep},j_{RC}} \tag{5.1}$$

5.2.1.4 Logarithm of the Modulus

LogMod is the process of computing the log of the energy of a signal, in this case of the output of Doppler Filtering.

This is a rather simple task, where the function $f(z) = 10 \log |z|^2$ (with domain $f : \mathbb{C} \to \mathbb{R}$) is applied to every sample. An optional stage of quantization can be applied at this stage, depending on the design of the radar, in order to switch to fixed-point arithmetics, featuring better performance and numerical stability at the cost of dynamic range.

5.2.1.5 Constant False Alarm Rate

Constant False Alarm Rate (CFAR) is a class of environment estimation method used to obtain a measure of the noise around a potential detection. This is used to normalize the output of the LogMod, and as such enabling the use of a fixed threshold while guaranteeing a constant false alarm rate.

Multiple CFAR methods exist, each featuring a different compromise between quality of noise estimation (sensitivity to jamming, clutter, etc), and execution time. We give a short list of implemented CFARs, although more variants exist.

In practice, a combination of CFARs is commonly used, in order to benefit from the advantages of each variation:

- Cell Averaging CFAR (CA-CFAR) is arguably the simplest kind of CFAR, as it relies on a kind of sliding mean[54]. It can be implemented as a simple 1D convolution, or with a custom kernel to avoid reading a filter from memory.
- Ordered-Statistic CFAR (OS-CFAR) may be the most robust existing CFAR. It relies on computing the n-th quantile of a window sliding over the signal[54].

5.2.1.6 Detection

Once the chosen set of CFARs has been evaluated, a heuristic is computed between the signal and the result of the CFAR methods, resulting in a normalized signal. The heuristic is chosen as a tradeoff between sensitivity and specificity.

This is the final step of the Signal Processing chain. Detections are then transferred to the next stage, where they are classified, clustered, and correlated with older detections in order to track objects on the long term.

5.2.2 Increasing the number of hypotheses

As described in Figure 5.4, only a few different hypotheses (the different branches on the figure) are made. This number is limited by the performance of the hardware executing the pipeline. When the number of hypotheses is low, each hypothesis must cover a large number of cases. Conversely, by increasing the number of hypotheses, we can design more narrow hypotheses, enabling finer resolution and higher gain throughout the chain. This helps detecting fainter targets, and giving more accurate estimates of the characteristics of a target.

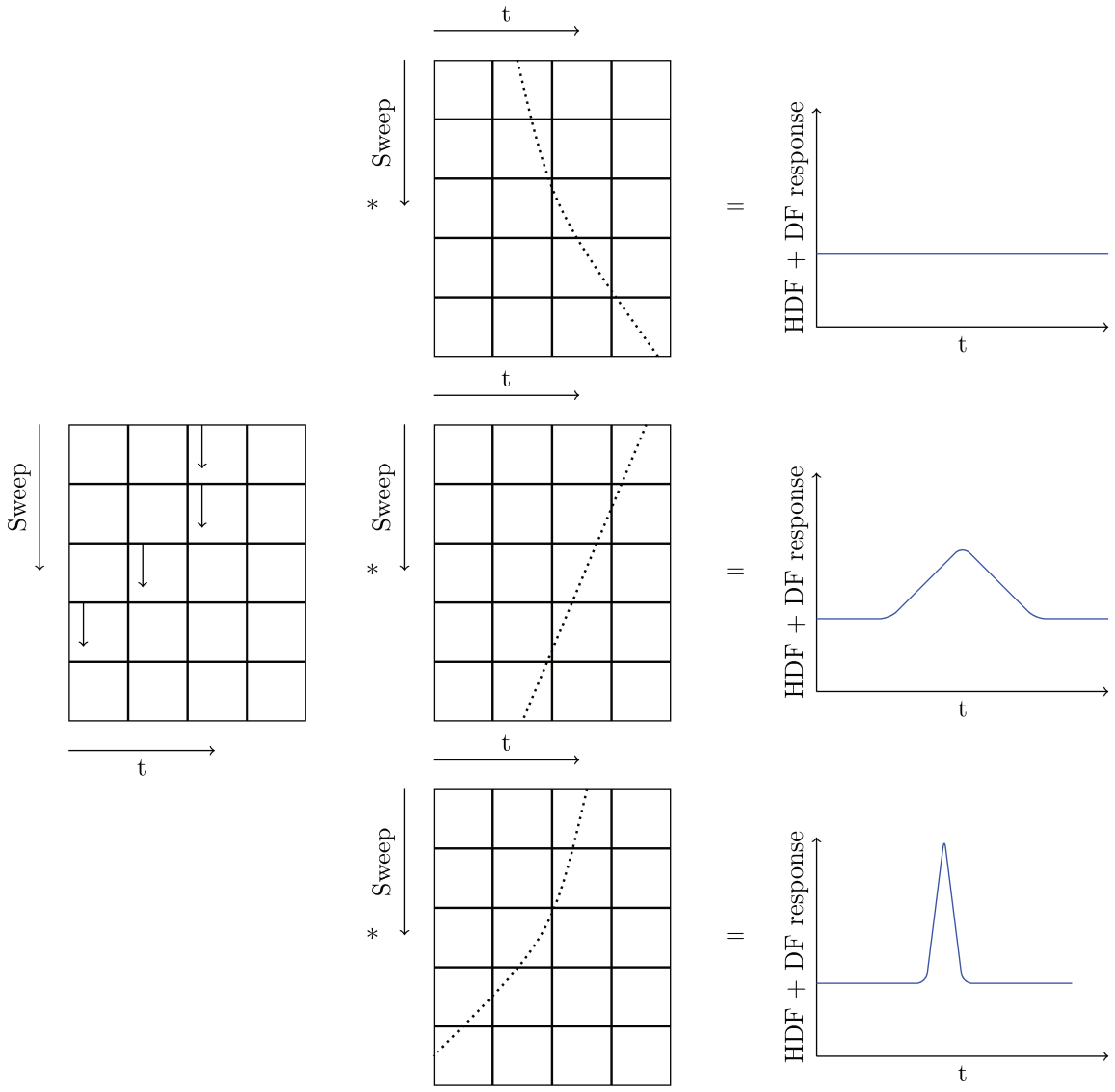


Figure 5.6: Matched filtering based on a filter bank in the context of Doppler Filtering

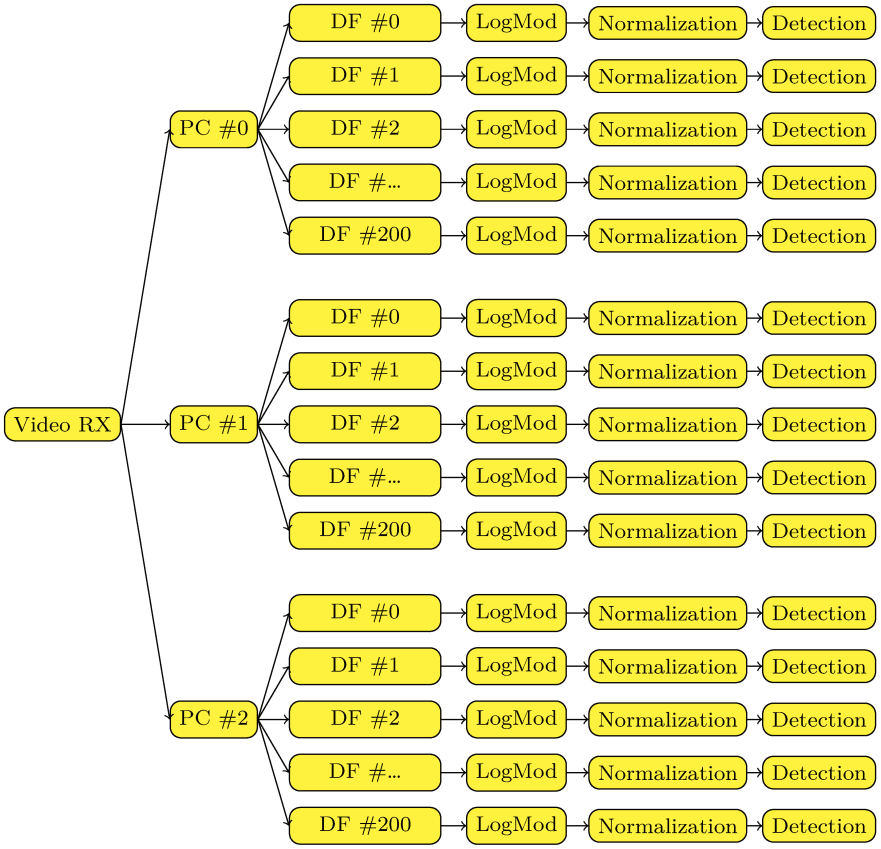


Figure 5.7: Massively Multi-Hypotheses Radar SP

5.2.2.1 Motivation

The data rate produced by radar antennas is continuously increasing, following the progression of Ethernet hardware, with the goal to detect always fainter and fast targets. This increasing data rate is becoming difficult to process using CPUs, leading to extreme systems, planning on using hundreds of multi-core CPU nodes.

At the same time, GPU computing is a revolution in the domain of numerical computing, bringing promises of huge improvements in terms of computation time and a potential solution to reduce the number of compute nodes, and at the same time reduce the energy consumption and cost of the computing systems. However, as explained throughout Section 5.1, blind rewriting of algorithms from CPU to GPU may lead to a poor GPU usage, and disappointing results.

By increasing the number of hypotheses, we make it possible to simplify greatly the processing chain, by using simple operations repeated at massive scale. This is exactly the kind of task at which GPUs excel, as it makes it possible to benefit as much as possible from their highly parallel architecture to process huge volumes of data.

Another advantage of increasing the number of hypotheses is to reduce straddling losses[18], which appear at each time a discretization is performed (different beams, hypotheses, …). By increasing the number of hypotheses, we can design more strongly overlapping filters, thus reducing straddling losses.

5.2.3 Implementation: `ConvSP`

We developed GPU implementations of these building blocks, in order to assess the feasibility of the Massively Multi-Hypotheses Radar SP (MMHRSP), identify potential bottlenecks, and provide a baseline to help the dimensioning of future GPU-based radars.

This work started from the assessment developed in Section 5.2.3.1, that many building blocks

of the MMHRSP are based on a single operation, the convolution, giving great optimization opportunities. This is what gave its name to this radar SP, `ConvSP`.

5.2.3.1 A common building block: the convolution

A basic assessment can be made for the three most compute-intensive building blocks of the primary radar SP (Sections 5.2.1.2, 5.2.1.3, and 5.2.1.5): Pulse Compression (PC), Doppler Filtering (DF) and CFAR (in the specific case of CA-CFAR) are in fact filters, which can themselves be implemented using 1D convolution.

The continuous 1D convolution of two functions f and g is given by:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(t)g(t-\tau)\,\mathrm{d}\tau$$

In the context of this work, we considered discrete signals exclusively. The practical expression for discrete 1D convolution that was used is given by:

$$\mathbf{y}_n = \sum_{s=1}^{S} \mathbf{x}_{n+s-\left\lfloor \frac{S}{2} \right\rfloor} \mathbf{h}_{S-s+1} \tag{5.2}$$

$$n \in [\![1, N]\!], \quad \{\mathbf{x}_k = P(\mathbf{x}, k) \mid k \in \mathbb{N} \setminus [\![1, N]\!]\} \tag{5.3}$$

$\mathbf{x} \in \mathbb{C}^N$ is the input signal, $\mathbf{h} \in \mathbb{C}^S$ is the filter, $\mathbf{y} \in \mathbb{C}^N$ is the result of the convolution, and $P : \mathbb{C}^N \times \mathbb{Z} \to \mathbb{C}$ is a padding function. Notice how the filter is centered over the input signal window, using $-\left\lfloor \frac{S}{2} \right\rfloor$. This avoids an undesirable offset in the result. The most common padding function used is simply $P(\mathbf{x}, k) = 0$, but others can be interesting to use as means to reduce edge effects.

5.2.3.2 Direct convolution

The straightforward computation of Equation 5.2 has a complexity of $\mathcal{O}(N^2)$. Albeit algorithms with lower complexity exist as shown in Sections 5.2.3.3 and 5.2.3.4, direct convolution is still competitive in terms of computation time for relatively short input vectors.

5.2.3.2.1 Naive kernel The most naive kernel maps one thread per output value of the convolution, working in an isolated fashion. Considering a filter with S components, each thread perform $2S$ memory reads, S Fused Multiply Add (FMA), and 1 memory write.

Memory accesses can be coalesced easily and a good proportion of cache hits can be achieved. As such, the performance of the naive kernel can be considered as rather good, despite the lack of complex optimizations.

By applying the method presented in Section 4.2, we provide a model of the execution time for this implementation on a few GPUs in Table 5.5.

5.2.3.2.2 Use of shared memory Shared memory usage is a common optimization in CUDA. This special memory subunit, presented in Section C.1.3, has much faster access than the global memory, and can be used to store frequently accessed elements.

However, in the case of convolution, the need for overlap makes shared memory usage very complicated, especially for larger filters. Implementations were written based on shared memory, and exhibit better performance than naive kernels. From the execution time model (Table 5.5), we can see that this implementation is roughly twice as fast as the naive method.

5.2.3.2.3 cuDNN CUDA DNN (cuDNN) is a library of the CUDA toolkit, a collection of high-quality libraries featuring GPU-accelerated primitives supported by Nvidia. cuDNN is under active development, as it is a backend for many AI libraries, such as Tensorflow and Pytorch. One of the most successful neural network of the last decade is the Convolutional Neural Network (CNN), based on the convolution operation. Because of this, cuDNN implements convolution, and arguably offers the best implementations in the domain of AI.

However, convolutions in the domain of AI are mostly 2D convolutions, with rather small filters (3x3, 5x5, ..., and up to 17x17). This is very different from our requirements, as we need 1D convolution, with possibly very wide filters (>1000 elements).

cuDNN usage is a good solution for quick prototyping, but fails to achieve more than ∼10 % of a GPU's peak performance.

5.2.3.2.4 CUTLASS Similarly to cuDNN, the CUTLASS library provides high performance primitives for linear algebra and 2D convolution, in a much more flexible way than cuDNN through the use of templates, and with first-class support for Tensor Core execution. It is described extensively in paragraph 5.2.5.2.4.

This library especially provides high-performance 2D convolution implementations, comparable with cuDNN in terms of performance. However, their implementations are slightly more restrictive than cuDNN's, and it is not possible to use these primitives with a size of 1 on one of the filter's dimensions, to effectively fall back to a 1D convolution.

However, this library was used at a lower abstraction level to provide a very high-performance implementation of DF targeting Tensor Cores.

5.2.3.3 FFT convolution

The classical convolution algorithm has an asymptotical complexity of $\mathcal{O}(n^2)$. By relying on the Fast Fourier Transform (FFT), we can reduce this complexity to $\mathcal{O}(n \log n)$. Indeed, the convolution theorem states that:

$$\mathcal{F}(x * y) = \mathcal{F}(x) \cdot \mathcal{F}(y) \tag{5.4}$$

On wide inputs, the overhead of computing the FFT is mitigated by the lower asymptotical complexity, making this solution very interesting in our context. However, because of the periodic property of the FFT, it is necessary to use enough zero padding in the input signal and filter before applying the transform in order to obtain the correct result. The size with padding of both the input signal and the filter must be greater or equal to $N_x + N_h - 1$. In case of mismatch between the signal size and the filter size, this can result in more computations than the direct convolution computation.

5.2.3.3.1 cuFFT The CUDA FFT (cuFFT) library is another part of the CUDA Toolkit. It provides highly optimized FFT implementations for both single GPU targets but also multi GPU FFTs.

It is the logical way to implement FFTs in CUDA, featuring good support and performance.

The only missing step to complete the FFT convolution was data padding and unpadding, as well as multiplication of the Fourier transformed input signal and filter. The latter can be done in a straightforward way using a naive CUDA kernel, but the padding operation is more tricky.

Indeed, the naive approach would be to allocate six intermediate arrays:

1. Padded input signal
2. Padded filter
3. FFT of input signal
4. FFT of filter
5. Multiplication of 3. and 4. (FFT of output signal)
6. Padded output signal

First, this is a lot of additional memory footprint, and it also requires many additional memory copies, to pad and unpad the different signals, which becomes a costly operation.

However, we relied on a feature of cuFFT, called cuFFT callbacks, making it possible to add user-defined functions on data load and data store. By using this feature, we were able to get rid of $^2/_3$ of the intermediate arrays (1., 2., 5. and 6.). Indeed, data could be padded on the fly

(eliminating intermediate arrays 1. and 2.) during the forward FFT, then multiplied on the fly (removing 5.) and cropped (6.).

This made it possible to provide a much more efficient and usable version of convolution implementation based on FFT, still subject to the limitations mentioned at the start of this section.

5.2.3.4 Overlap-save method

Alternative methods exist to address these limitations and reduce the need to compute very long FFTs, the most popular being overlap-save and overlap-add methods.

In the context of this work, we focused on the overlap-save method, which is easier to implement on GPU as it does not require the use for atomic operations or specific synchronization, unlike the overlap-add.

5.2.3.4.1 Description These methods rely on the idea to compute convolutions on small segments of the input signal, thus reducing the required FFT size from $N_x + N_h - 1$ to $N'_x + N_h - 1$, where $N'_x << N_x$ is a segment of the input signal x[49].

5.2.3.4.2 Optimal FFT size The ratio between the number of computations required to perform FFT convolution using the naive method presented in Section 5.2.3.3 and the overlap-save method is given by[49]:

$$f(M, N) = \frac{N(\log_2(N) + 1)}{N - M + 1} \tag{5.5}$$

Where M is the filter size and N is the FFT size. This ratio can be used as a cost function in an optimization problem, in order to find the optimal value for the FFT size:

$$N^*(M) = \min_{N=2^k} f(M, N), \quad N \geq M \geq 2 \tag{5.6}$$

This minimization problem can be solved using a bruteforce method over a limited set of FFT sizes, but we performed additional analysis in order to find a closed form for this optimal size, additionally leading to valuable information: the corresponding inverse function, giving the optimal filter size corresponding to a specific FFT size, and the asymptotic evolution of the optimal FFT size.

In order to find a closed form for this problem, we started by finding the closed form solution of the simpler, continuous problem:

$$N^*(M) = \min_{N \in \mathbb{R}} f(M, N), \quad N \geq M \tag{5.7}$$

We begin by finding the derivative of Equation 5.6 with respect to N:

$$\frac{\partial}{\partial N} f(M, N) = -\frac{N(\log_2(N) + 1)}{(N - M + 1)^2} + \frac{\log_2(N) + 1 + \frac{1}{\log(2)}}{N - M + 1} \tag{5.8}$$

Since $N \geq M \geq 2$, the denominator is strictly positive, and $\log_2(N)$ is always defined and continuous. By composition, the derivative is continuous on its definition domain.

We proceed by finding the roots of Equation 5.8, potential extrema of f, as a function of M:

$$\frac{\partial}{\partial N} f(M, N) = 0$$
$$\frac{\log_2(N) + 1 + \frac{1}{\log(2)}}{N - M + 1} = \frac{N(\log_2(N) + 1)}{(N - M + 1)^2}$$

By multiplying both sides by $(N - M + 1)^2$, developing and reordering, we obtain:

$$N + (1 - M)(\log(N) + \log(2) + 1) = 0 \tag{5.9}$$
$$\frac{N}{1 - M} + \log(N) + \log(2) + 1 = 0$$

We begin to see an expression of the form $we^w = z$, which is solved by the Lambert W function.

$$2eNe^{N/1-M} = 1$$
$$\frac{N}{1 - M} e^{N/1-M} = \frac{1}{2e(1 - M)} \tag{5.10}$$

We can finally apply the Lambert W function:

$$N^* = (1 - M)W\left(\frac{1}{2e(1 - M)}\right) \tag{5.11}$$

However, it is defined over the complex plane, and possesses multiple branches denoted by integers k. The k-th branch is noted W_k. We need to choose the right branch, or we will end up with the wrong solution.

Firstly, $W(x)$ is real only if $x \in \left[-\frac{1}{e}, +\infty\right[$. Secondly, $ye^y = x$ is solved by $W_0(x)$ when $x \in \left[-\frac{1}{e}, +\infty\right[$, and additionally by $W_{-1}(x)$ when $x \in \left[-\frac{1}{e}, 0\right[$. To make sure that the Lambert W function can solve this problem, and to know which branch must be used, we must find bounds for the right-hand side of Equation 5.10

$$\begin{aligned} 2 \leq \quad & M & < +\infty \\ -1 \leq \quad & \frac{1}{1 - M} & < 0 \\ -\frac{1}{2e} \leq \quad & \frac{1}{2e(1 - M)} & < 0 \end{aligned}$$

On this interval I, both W_0 and W_{-1} are defined. In order to discriminate which branch to use, we can use the property that W_0 is strictly increasing on I, and that W_{-1} is strictly decreasing, as well as an identity of W, $W(z) = {}^{z}/_{e^{W(z)}}$. Starting back from Equation 5.11:

$$N^* = \frac{1}{2e^{W(1/2e(1-M))+1}} \tag{5.12}$$

Using this form allows to dodge an indeterminate form when multiplying the inegality by $(1 - M)$. We can now proceed with the image of $N*$ using W_0:

$$\frac{1}{2e} < N_0^* \leq \frac{1}{2e^{W(-1/2e)+1}}$$
$$0.18394\cdots < N_0^* \leq 0.23196\ldots$$

This solution is irrelevant, as $N \geq 2$. The proper branch must be W_{-1}, we can verify it by calculating the image of N^* using W_{-1}:

$$\frac{1}{2e^{W(-1/2e)+1}} \leq N_{-1}^* < \lim_{M \to +\infty} \frac{1}{2e^{W(-1/2e(1-M))+1}}$$
$$2.67835\cdots \leq N_{-1}^* < +\infty$$

Indeed, the image of N^* when using W_{-1} matches the constraints of our solution. The only root of Equation 5.8 and extremum of Equation 5.5 on our definition domain is given by:

$$N^*(M) = (1 - M)W_{-1}\left(\frac{1}{2e(1 - M)}\right) \tag{5.13}$$

The last step is to check if this point is a maximum or a minimum, or an inflection point. This can be verified by determining the sign of the derivative around this point.

$$\frac{\partial}{\partial N} f(M,M) = (1-M)(\log_2(M)+1) + \frac{1}{\log(2)}$$
$$\lim_{N\to+\infty} \frac{\partial}{\partial N} f(M,N) = +\infty$$

By performing the same kind of analysis for the first member of this set of equations, omitted for brevity, we found that the image of $\frac{\partial}{\partial N} f(M,M)$ on $I = [2, +\infty[$ is $]-\infty, \frac{1}{\log(2)} - 2]$, which is strictly negative.

Since f is decreasing before N^*, and increasing after it, N^* is a minimum of f. Since it is the only value of N for which $\frac{\partial}{\partial N} f(M,N) = 0$ on the definition domain of f, N^* is the global minimum of f.

Finally, we can give the optimal FFT size solving Equation 5.6. Figure 5.8 shows the first values of the optimal value.

$$\boxed{N(M) = 2^{\lfloor \log_2(N^*(M)) \rceil}} \tag{5.14}$$

5.2.3.4.3 Approximation During the initial bruteforce study, and after plotting N^* such as on Figure 5.8, we wondered if a linear approximation existed for the optimal value N^*. We performed linear regressions on the bruteforced plateaus found for N, without success.

However, starting from the closed form solution over reals Equation 5.13, we can easily produce approximations.

Indeed, the Lambert W function can be expressed and approximated using a recursion. Starting from one of the function's elementary property:

$$\begin{aligned} W(x)e^{W(x)} &= x \\ \log(-W(x)e^{W(x)}) &= \log(-x) \\ W(x) &= \log(-x) - \log(-W(x)) \end{aligned} \tag{5.15}$$

This expression can be injected in Equation 5.13. We denote by N_n^* the expression of N^* approximated by a recursion formula of depth n.

The first few N_n^* are given by:

$$\begin{aligned} N_0^* &= M - 1 \\ N_1^* &= (M-1)(\log(M-1) + \log(2) + 1) \\ N_2^* &= (M-1)(\log(M-1) + \log(2) + 1 + \log(\log(M-1) + \log(2) + 1)) \end{aligned}$$

This representation, in spite of being linear for $n > 0$, is very simple to implement (W is not defined in the `C++` standard library), and a depth of $n \geq 2$ is arguably enough to approximate N^* with enough accuracy. It plays a central role in the setup of our overlap-and-save implementation, as it dictates which FFT size must be used.

However, we were not able to find a relation with commonly used approximation to this optimal size, such as $N = \alpha M$, where α is typically chosen as 8 or 10[49]. Indeed, even if this approximation might hold for medium M, it is wrong both for small and large values. Moreover, N^* is not asymptotically linear:

$$\alpha \in \mathbb{R}^{+*}, \beta \in \mathbb{R}, \quad \lim_{M\to+\infty} \frac{N^*}{\alpha M + \beta} = +\infty$$

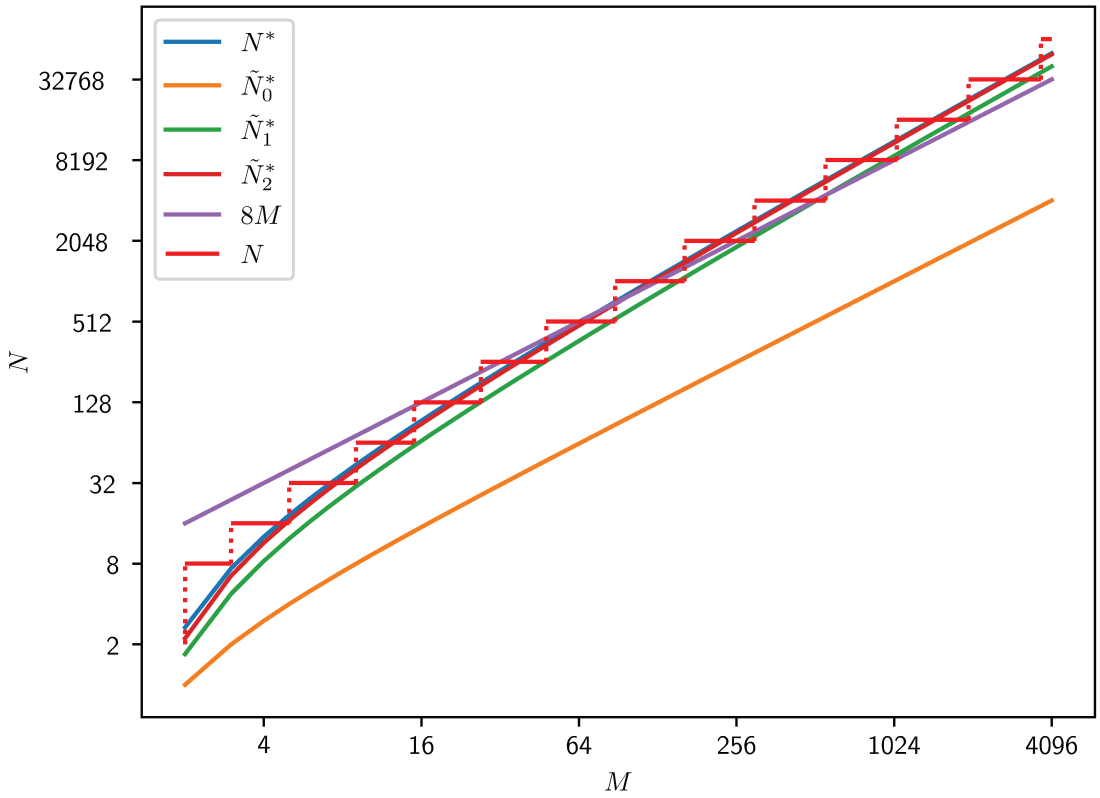


Figure 5.8: Optimal FFT size N as a function of filter size M and approximations for the overlap-and-save method

Compute Capability	FFT Size Range N	Filter Size Range M
75	2–4096	2–560
70, 72, 86 and 89	2–16 384	2–1973
80, 87 and 90	2–32 768	2–3727

Table 5.4: cuFFTDx supported FFT sizes for FP32 Complex-to-Complex (C2C). Note that the overlap-save method can still be used up to $N = M$, however the overlap-save method will be suboptimal.

5.2.3.4.4 Inverse It is also desirable to be able to find the filter size M associated with a specific FFT size. For example, the library cuFFT Device eXtension (cuFFTDx) supports a limited set of FFT sizes, such an inverse function can be used to understand the maximum filter size of a convolution implemented with the overlap-and-save method and cuFFTDx.

Starting back from Equation 5.9, and solving for M:

$$M(N) = \frac{N}{\log(N) + \log(2) + 1} + 1 \tag{5.16}$$

More work would be needed to express this inverse function as a multi-valued function, inverse of Equation 5.14. This would give interesting insights on the range of values covered by a single FFT size.

5.2.3.4.5 Implementation with cuFFTDx cuFFTDx is a library providing FFT implementations as device functions. This makes it possible to mix FFTs with other kernel code, and achieve kernel fusion in a much flexible way than using cuFFT callbacks.

The implementation with cuFFTDx was very straightforward once the optimal FFT size was known. All filters are transformed using the FFT at initialization, in order to avoid repetitive, unnecessary computations.

Chunks of input signal are then loaded to shared memory with appropriate padding, FFT'ed, multiplied with said signals in Fourier space, and transformed back to time domain, all in one go: data does not need to be stored back in global memory at each stage, in hopes of great significant improvement compared to the cuFFT approach. This can limit greatly the set of filter sizes supported.

A drawback of this implementation is that the input signal is not necessarily a multiple of the optimal FFT size. In case of a large mismatch this can lead to a lot of unnecessary computations. This effect is amplified for larger filter sizes, as the optimal FFT size follows a kind of quasilinear trend. A workaround would be to derive the optimal FFT size for FFTs of other sizes than powers of two.

The second drawback of this implementation is that it is dependant on the set of FFTs sizes supported by cuFFTDx. This set is currently dependant on the GPU's Compute Capability and given by Table 5.4.

Thanks to the use of the FFT, the execution time of this implementation is theoretically logarithmic as a function of the filter width S. This makes this implementation very efficient, however it is more difficult to compare to the other implementations, linear in all parameters, where a simple comparison of the regression coefficient is enough to assess which implementation is faster.

Interestingly enough, the execution time first increases, and then decreases for successive filter sizes, as shown on Figure 5.9. This may be a consequence of more input samples being processed at once on bigger filter sizes, resulting in better memory coalescence, or an implementation detail in cuFFTDx. Plateaus inside the same color zone (same FFT length) are expected to take roughly the same time, as roughly the same amount of computations is required, with the exception to edge effects. However, plateaus spanning multiple color zones are unexpected. We did find a model matching this evolution of the execution time.

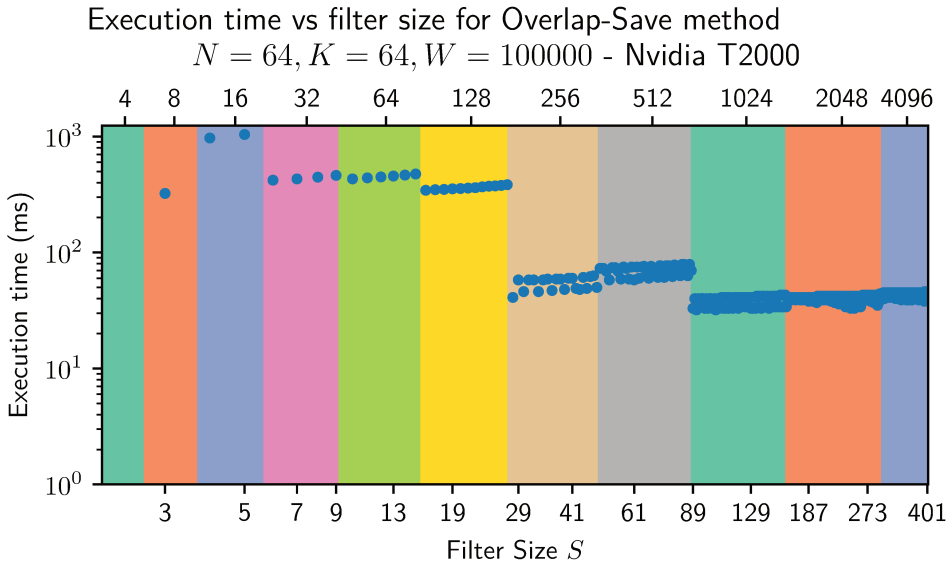


Figure 5.9: Execution time *vs* filter size.

5.2.3.5 Results

These different implementations were benchmarked using the method presented in Section 4.2. All of them were showed to be linear with respect to the batch size N, the number of filters K and the input signal length W, making it possible to explore our complete design space while plotting relative execution times. The results obtained using this method are plotted in Figure 5.10, and a summary of the models produced is given in Table 5.5.

We notice that except for the smallest filter sizes, the overlap-save method outperforms other implementations. Additionally, the direct FFT approach shows similar execution time compared to cuDNN, however with a different evolution. More benchmarking will be required in order to find the minimum filter size required in order to make this method worth using.

Implementation	Model	R^2	Uncertainty
Naive	$1.717 \times 10^{-9} \times N \times K \times W \times S\,\text{ms}$	1	1.231×10^{-13}
Shared memory	$8.957 \times 10^{-10} \times N \times K \times W \times S\,\text{ms}$	0.9999	3.189×10^{-13}
cuDNN	$4.822 \times 10^{-9} \times N \times K \times W \times S\,\text{ms}$	1	5.646×10^{-13}
FFT (cuFFT)	N/A	N/A	N/A
FFT (overlap-save)	N/A	N/A	N/A

Table 5.5: Execution time models for convolution implementations. No model could be found for the overlap-save method.

5.2.4 PC implementation

Pulse Compression (PC) is a simple filtering operation, implemented directly with a 1D convolution, without any further step. Because of this, any convolution implementation presented above can be used.

The peculiarity of PC is that filters are relatively long, generally about 15 % of the signal size, resulting in filter sizes typically in the range $[100, 100\,000]$. Because of these long filters, the overlap-save method is generally the best choice to implement PC.

5.2.5 DF implementation

Doppler Filtering (DF), however, is not a straightforward 1D convolution operation. It can be seen either as 2D convolution where filter height matches signal height, resulting in no sweeping

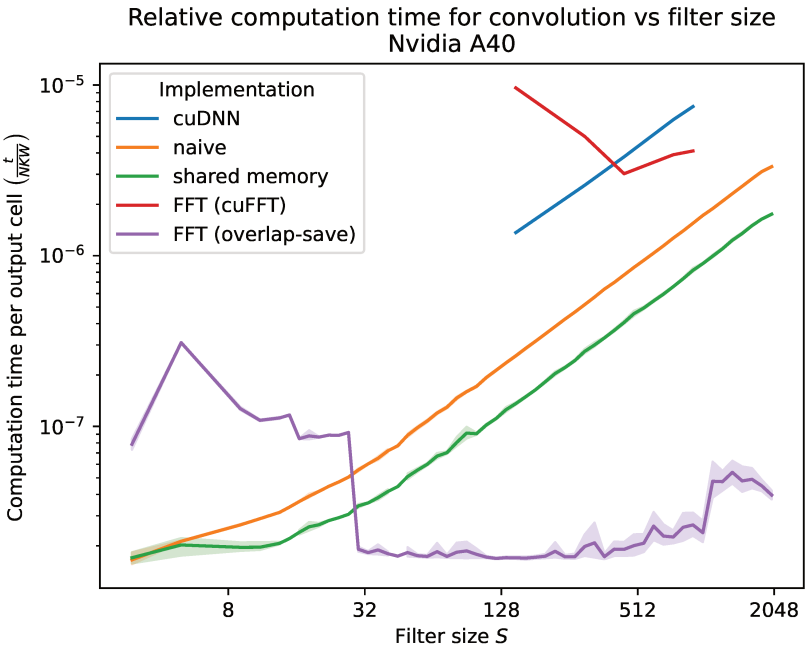


Figure 5.10: Relative execution time evolution as of filter size for convolution implementations

on the vertical axis. It can also be seen as a set of 1D convolutions, followed by a reduction over the vertical axis. Finally, we also created a method relying on the sparsity of DF filters to reduce the execution time.

In this section, we expand on the different implementations developed for this stage.

5.2.5.1 Dense DF

The first, most naive, implementation is the dense DF, the direct implementation of the 2D convolution as presented in Equation 5.1.

5.2.5.1.1 cuDNN As mentioned earlier in this thesis, cuDNN is specialized for 2D convolution. Because of this, it seems logical to leverage this approach in order to use cuDNN in the most efficient way.

This is done in a straightforward way, with the exception that cuDNN does not support operation on complex numbers, even though Doppler Filtering has both complex input and output.

In order to actually implement complex multiplication based on a real interface, we took advantage of the expression of the convolution as computed by cuDNN, slightly expanded compared to Equation 5.2:

$$y_{n,k,p,q} = \sum_{c}^{C} \sum_{r}^{R} \sum_{s}^{S} x_{n,c,p+r,q+s} \times h_{k,c,R-r-1,S-s-1} \tag{5.17}$$

This expression takes into account the second axis of convolution (traversed by variable r), as well as the multiple channels (c), and the batch size (n). Notice how the convolution of each different input channel is summed together. This specific feature, deemed unintuitive at first, was actually relied upon to implement complex multiplication.

By storing real part at index $c = 0$ and imaginary part at index $c = 1$ and cleverly duplicating the filter, we were able to implement complex multiplication over the real convolution interface of cuDNN.

More precisely, we want to compute the complex product between the input signal element x and the filter element h. This product can be expanded, and assigned to specific elements of cuDNN's NCHW/KCRS layouts:

$$x_i h_j = (\Re(x) + i\Im(x))(\Re(h) + \Im(h)) = \underbrace{(\overbrace{\Re(x)\Re(h)}^{c_0} - \overbrace{\Im(x)\Im(h)}^{c_1})}_{k_0} + i\underbrace{(\overbrace{\Re(x)\Im(h)}^{c_0} + \overbrace{\Im(x)\Re(h)}^{c_1})}_{k_1}$$

Under matrix form:

$$x \cdot h = k \downarrow \overbrace{\begin{pmatrix} \Re(h) & -\Im(h) \\ \Im(h) & \Re(h) \end{pmatrix}}^{c} \cdot c \downarrow \begin{pmatrix} \Re(x) \\ \Im(x) \end{pmatrix} = k \downarrow \begin{pmatrix} \Re(x)\Re(h) - \Im(x)\Im(h) \\ \Re(x)\Im(h) + \Im(x)\Re(h) \end{pmatrix}$$

Based on this representation, it is apparent that by duplicating carefully the complex filter, we can implement the complex convolution using the real convolution interface proposed by cuDNN. Note that this trick could be reversed, duplicating the input signal. However, filter banks are most of the time available at initialization time, and can be duplicated once, before being reused multiple times.

5.2.5.1.2 DF rewritten as a matrix multiplication

By swapping the sums in Eq. Equation 5.1, the DF operation becomes the sum of matrix multiplications:

$$DF = \sum_{j_{RC}=-\lfloor \frac{S}{2} \rfloor}^{\lfloor \frac{S}{2} \rfloor} H_{j_{RC}} \cdot (IQ \cdot L^{j_{RC}}) \tag{5.18}$$

Where:

- H is a $H \times (N_{HDF} N_{DF}) \times N_{Sweep}$ tensor, a concatenation of the $h_{i_{HDF}, i_{DF}, i_{Sweep}, j_{RC}}$ from Equation 5.1
- IQ is the input signal as a $N_{Sweep} \times N_{RC}$ matrix
- $L^{j_{RC}}$ is the j_{RC}-th lower shift matrix given by $L^{j_{RC}}_{i,j} = \delta_{i,j+j_{RC}}$, where $\delta_{i,j}$ is the Kronecker delta function

GPUs are especially designed to perform matrix multiplication efficiently, so this approach is very promising. However, we only applied it combined to another promising optimization, the Sparse DF (see paragraph 5.2.5.2.4).

5.2.5.2 Sparse DF

At higher sampling frequencies, the cell migration effect increases, as each cell represents a shorter time. Because of this, the Doppler filter becomes more and more sparse, as shown on Figure 5.11.

A typical Doppler filter features sparsity in a non standard format: each row first contain a number of zeros, then a dense array of coefficients, and zeros again. This format can be described very efficiently using the offset of the first non-zero element per row, and the number of non-zero elements per row. An illustration of this method is given in Figure 5.12.

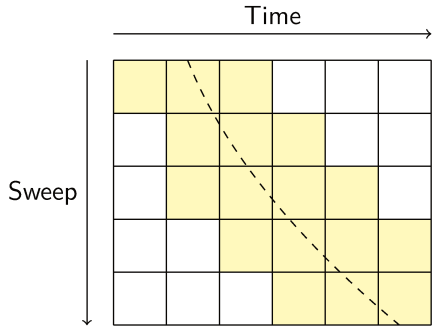
5.2.5.2.1 Mathematical expression

Using this representation, the expression of Doppler Filtering Equation 5.1 becomes that of Sparse Doppler Filtering:

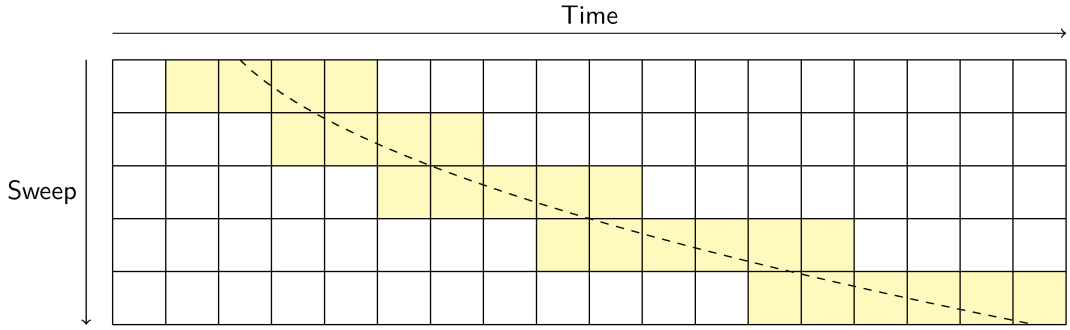
$$DF_{i_{HDF}, i_{DF}, i_{RC}} = \sum_{i_{Sweep}=0}^{N_{Sweep}-1} \sum_{j_{RC}=-\lfloor \frac{s_{i_{HDF}, i_{Sweep}}}{2} \rfloor}^{\lfloor \frac{s_{i_{HDF}, i_{Sweep}}}{2} \rfloor} IQ_{i_{Sweep}, (i_{RC} + \Delta_{i_{HDF}, i_{Sweep}} + j_{RC})} \cdot h_{i_{HDF}, i_{DF}, i_{Sweep}, \Delta_{i_{HDF}, i_{Sweep}} + j_{RC}} \tag{5.19}$$

Where we introduce two additional terms:

- Δ is the offset for a specific Sparse Doppler Filter and Sweep



(a) Low sampling rate



(b) High sampling rate

Figure 5.11: Dense representation of a Doppler filter at low and high sampling frequency. Yellow cells represent non-zero values.

- s is the number of non-zero elements per row

It is also interesting to establish an expression to find the compression ratio enabled by this method on the filter h. The compression ratio is given by:

$$\text{Compression Ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}$$

.

In the context of DF:

- Uncompressed Size = $N_{HDF} N_{DF} N_{Sweep} S \cdot \texttt{sizeof(T)}$
- Compressed Size = $N_{HDF} N_{DF} N_{Sweep} \left(\max s \cdot \texttt{sizeof(T)} + 2 \cdot \texttt{sizeof(IndexT)}\right)$

As such, the general formula giving the compression ratio is:

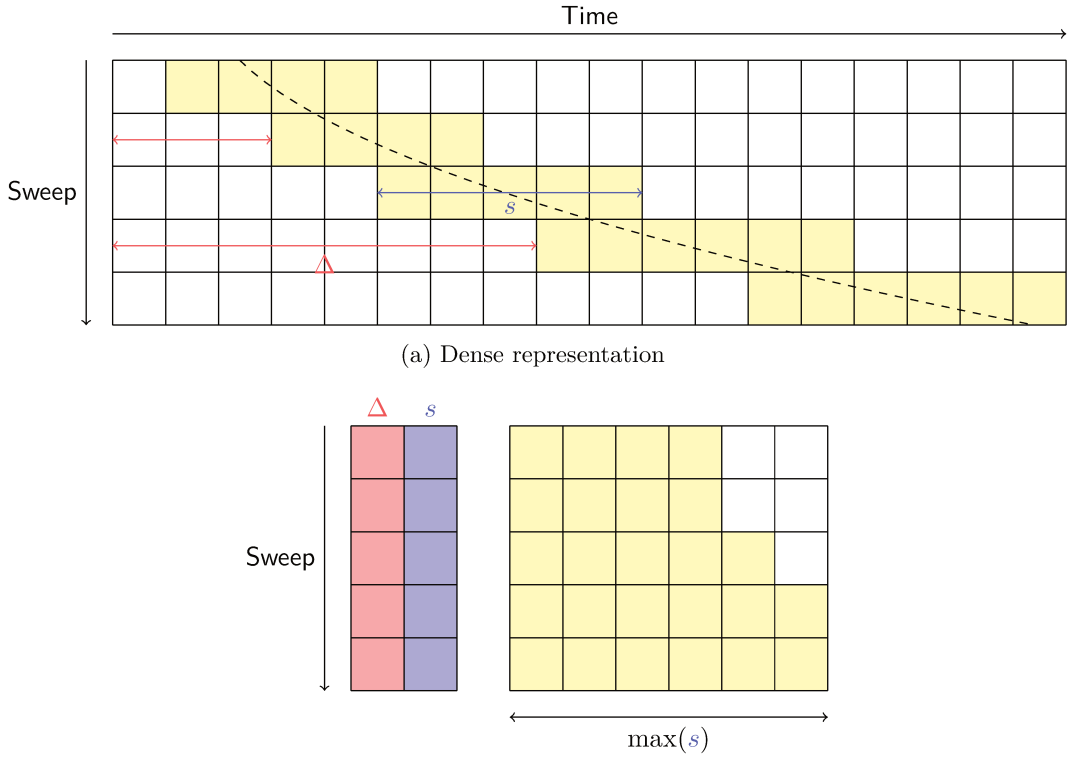
$$\text{Compression Ratio} = \frac{S \cdot \texttt{sizeof(T)}}{\max s \cdot \texttt{sizeof(T)} + 2 \cdot \texttt{sizeof(IndexT)}} \tag{5.20}$$

Where `T` is the data type of the filter's components, and `IndexT` is the data type used to specify Δ and s. The typical choice for `T` is a pair of FP32, representing a complex number, of size 8 B; regarding `IndexT`, the default is UINT32, of size 4 B. `sizeof` is the function returning the number of bytes needed to store the data type passed as argument.

In this specific case, the compression ratio can be simplified:

$$\text{Compression Ratio}\big|_{\texttt{sizeof(T)}=8,\texttt{sizeof(IndexT)}=4} = \frac{S}{\max s + 1}$$

Logically, the compression ratio increases for lower values of $\max s$, the maximum number of non-zero elements per row across all filters. Taking the global maximum value of s is a strong limitation: s can be very different from one Doppler Filter Hypothesis (HDF) to another. In this



(a) Dense representation

(b) Compressed representation

Figure 5.12: Compressed representation of a sparse Doppler filter

case, the compression ratio is bounded by the extreme HDFs, featuring large s, reducing the overall achievable compression.

However, a variable size for h's rows would be much more difficult to implement, and could lead to decreased performance due to the need of more complicated pointer arithmetic, or a double pointer dereference.

A proposed workaround is to split the DF stage in multiple kernel runs, grouping hypotheses with similar s together. Batching is still desirable, and every HDF should not be launched separately, as the GPU's resources might not be fully used doing so.

In practice, the compression ratio lies in the range 5–10, enabling significant performance improvements. Indeed, even though the memory footprint of storing the filter is not limiting in itself, the time to transfer the filter from global memory to registers is a huge performance bottleneck.

5.2.5.2.2 Naive kernel The naive kernel approach is very similar to that presented for 1D convolution in paragraph 5.2.3.2.1. It relies on caching to implement 1D convolution of the compressed filters efficiently, and accumulates the result from different sweeps in register, to limit data movement.

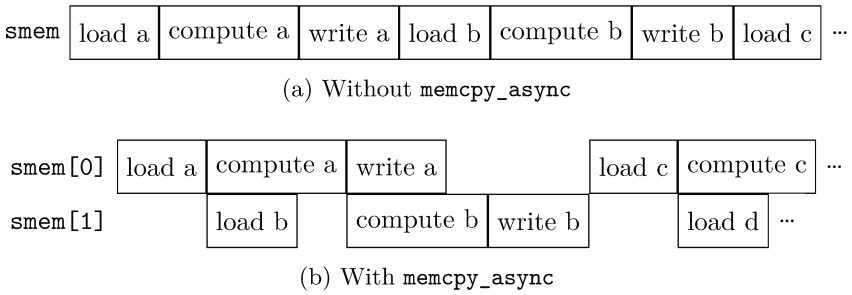
This version already provides acceptable performance, but with compressed filters, it can become interesting and more feasible to use shared memory, as described in the next section.

5.2.5.2.3 Shared memory + `memcpy_async` The compute latency of the convolution can be hidden by using `memcpy_async`[48]. This function is supported by GPUs with Compute Capability higher or equal to 5.0, and is actually asynchronous, with special hardware support, on devices with compute capability of 8.0 or more.

When compiling to a GPU with the right hardware support, this function maps to a specific instruction (`cp.async`), performing an asynchronous memory transaction between global memory and shared memory.

This mechanism induces additional complexity to avoid race conditions, but can reduce the execution time of a kernel from $t = t_{Compute} + t_{Memory}$ up to a lower bound of $t = \max(t_{Compute}, t_{Memory})$, where $t_{Compute}$ and t_{Memory} respectively denote the execution time spent on Compute and Memory operations.

To avoid race conditions, a double buffering system is implemented in shared memory, as represented on Figure 5.13.



(a) Without `memcpy_async`

(b) With `memcpy_async`

Figure 5.13: Timelines of execution of two kernels, with and without `memcpy_async`

5.2.5.2.4 CUTLASS CUTLASS is an open-source, header-only, linear algebra library distributed by Nvidia for CUDA. It shows similar results in terms of performance than CUDA BLAS (cuBLAS) and cuDNN, the respective references in the domains of matrix multiplication and 2D convolution, in a much more configurable and reusable way through the use of templates.

It is the preferred way to access the GPU's Tensor Cores, a special arithmetic unit specialized for the Matrix Multiply Accumulate (MMA) operation, that can deliver up to 4 times more TFLOP/s than regular CUDA cores, if used properly. The MMA operation is $D = AB + C$, where $A \in \mathcal{M}_{M,K}(\mathbb{K})$, $B \in \mathcal{M}_{K,N}(\mathbb{K})$, $\{C, D\} \in (\mathcal{M}_{M,N}(\mathbb{K}))^2$, and $\mathbb{K} \in \mathbb{Z}, \mathbb{R}$. In terms of actual data types, MMA is available for a varying set of data and matrices sizes, depending on the Compute Capability. The main choices of data types are:

- FP16/32/64
- (U)INT4/8/16/32/64
- BF16, a special 16-bit floating point format by Nvidia with greater dynamic range but less precision than FP16
- TF32, a special 19-bit floating point format by Nvidia with same dynamic range than FP32 and BF16, and intermediate precision between FP32 and BF16.
- BOOL, with a choice of operator: AND or XOR.

Tensor Core usage is very complex, as they require a very specific arrangement of the matrices A, B, C and D in memory. The matrices are distributed among threads of a warp as fragments. For example, in the case of the m16n8k4 MMA with TF32 operands and F32 accumulator, the matrix A must be distributed as shown on Table 5.6.

In this example, each thread instantiates a static array of two TF32 elements, a_0 and a_1. Thread 0 holds the value of $A_{0,0}$ in a_0 and $A_{8,0}$ in a_1, thread 1 hold $A_{0,1}$ and $A_{8,1}$, and so on.

Arrangements for B, C and D are different from this one, although C and D share the same arrangement.

This arrangement is very complex, some configurations, such as FP16 m8n8k4 perform 4 MMA per warp, in a single instruction.

Finally, as a last evidence of the complexity of the MMA family of instructions, similar instructions exist, with yet different constraints and arrangements: the Warp MMA (WMMA), featuring larger matrix sizes, and the WarpGroup MMA (WGMMA), performing an asynchronous MMA across multiple workgroups starting with the new Hopper architecture (CC 9.0).

Row\Col	0	1	2	3
0	T0:a_0	T1:a_0	T2:a_0	T3:a_0
1	T4:a_0	T5:a_0	T6:a_0	T7:a_0
...				
7	T28:a_0	T29:a_0	T30:a_0	T31:a_0
8	T0:a_1	T1:a_1	T2:a_1	T3:a_1
9	T4:a_1	T5:a_1	T6:a_1	T7:a_1
...				
15	T28:a_1	T29:a_1	T30:a_1	T31:a_1

Table 5.6: Arrangement of matrix A on a TF32 m16n8k4 MMA

Level	Description
Device	Callable from host, similar to cuBLAS's `gemm`
Kernel	Full kernel as a `__device__` function. Can be used to perform kernel fusion at a global scale. Memory loads and stores are still abstracted
CUDA block	Operation distributed across a full CUDA block. Memory loads and stores must be done separately, allowing great flexibility but more complexity
Warp	Abstract MMA at warp scale
Thread	MMA in a single CUDA core (alternative to Tensor Cores).
Instruction	`mma`, `wmma`, `wgmma`, ...

Table 5.7: Abstraction levels for CUTLASS' GEMM API

All of these technical details lead Nvidia to provide much higher-level interfaces to Tensor Cores, since keeping up with all of these variants both during development time and maintenance time requires significant effort.

The first, and simplest means to benefit from the performance gain brought by Tensor Cores is to use CUDA toolkit libraries supporting Tensor Cores, such as cuBLAS or cuDNN. These libraries seamlessly make use of Tensor Cores if possible. The first drawback of this approach is very little control on Tensor Core usage, as there is generally no way to force these libraries to use Tensor Cores (only hints are available, but a silent fallback on regular CUDA cores is always possible). The second drawback is a lack of flexibility, as it requires our problem to fit in the rather small interface provided by these libraries. Kernel fusion is limited if possible, and non-standard memory arrangements (such as ring buffers or matrices with each row featuring a specific offset, two arrangements we used in the context of this PhD) are impossible to use.

The second interface to Tensor Cores is the CUTLASS library, providing multiple levels of abstraction, in order to let the user choose which level is the best. The different levels are summarized in Table 5.7, and concern the CUTLASS' GEneral Matrix Multiply (GEMM) API. The 2D convolution API may be different, and only the device-level abstraction is documented.

Because of this promise of good performance and flexibility, we implemented the DF using CUTLASS, based on equation Equation 5.18.

We initially planned to use CUDA block-level abstraction. However, because of the complexity of CUTLASS' code, and its Work-in-Progress (WIP) documentation, we did not succeed to modify data load functions to match the specifics of the Sparse DF.

However, we did manage to use the instruction-level interface, to implement a valid DF. Nevertheless, the implementation is only valid with one instruction, and the convoluted repartition presented in Table 5.6 is manually implemented.

We also implement manually the complex convolution in a clever way that could be proposed to CUTLASS at some point.

5.2.5.2.4.1 Architecture We applied Equation 5.18, modified to match the Sparse DF algorithm:

$$DF_{i_{HDF}} = \sum_{-\lfloor \frac{\max s}{2} \rfloor}^{\frac{\max s}{2}} \widehat{H}_{i_{HDF},j_{RC}} \cdot \widehat{L}_{i_{HDF},j_{RC}} (IQ) \tag{5.21}$$

$\widehat{H}$ is the sparse version, as described in Section 5.2.5.2, of the H matrix introduced in Equation 5.18. $\widehat{L}$ is a non-linear shift operator, as in the sparse DF, each row of IQ is shifted by a different factor, $\Delta_{i_{HDF},i_{Sweep}}$ (*cf.* Equation 5.1).

5.2.5.2.4.2 Complex matrix multiplication The classical complex multiplication algorithm (as opposed to the variant using only 3 multiplications[33, p. 706]) of two complex numbers $z_1 = a_1 + ib_1$ and $z_2 = a_2 + ib_2$ is given by:

$$z_1 z_2 = (a_1 a_2 - b_1 b_2) + i(a_1 b_2 + b_1 a_2) \tag{5.22}$$

In CUTLASS' abstractions, the chosen method to perform the complex MMA of size (M, N, K) is to use 4 different real MMAs of the same size. This approach was not suited to our data layout, as it benefits from an NCHW layout. Even if supporting more modes is better, the initial focus for DF was an NHWC layout. Additionally, the granularity of the MMA was not ideal for radar signals: to achieve maximal performance, the dimensions of the DF must be a multiple of (M, N, K).

Instead, we managed to realize the complex MMA of size $(\frac{M}{2}, \frac{N}{2}, K)$ using one real MMA. This provides both smaller granularity and a more NHWC-friendly algorithm, since whole complex numbers are processed at once, and not in multiple steps.

In order to do so, we arranged the matrices in block matrices:

$$\tilde{A} = \left(\begin{array}{c} \Re(A) \\ \hline \Im(A) \end{array}\right), \tilde{B} = \left(\begin{array}{c|c} \Re(A) & \Im(A) \end{array}\right), \tilde{D} = \tilde{A}\tilde{B} = \left(\begin{array}{c|c} \Re(A)\Re(B) & \Re(A)\Im(B) \\ \hline \Im(A)\Re(B) & \Re(A)\Im(B) \end{array}\right) \tag{5.23}$$

It is then a simple matter of reducing $\tilde{D}$ to D according to equation Equation 5.22:

$$D = (\tilde{D}_{1,1} - \tilde{D}_{2,2}) + i(\tilde{D}_{1,2} + \tilde{D}_{2,1}) \tag{5.24}$$

The C term of the MMA can either be added in equation Equation 5.24 or directly in equation Equation 5.23, expanded as the real matrix $\tilde{C}$. This expansion is not unique, the most trivial probably being the following:

$$\tilde{C} = \left(\begin{array}{c|c} \Re(C) & \Im(C) \\ \hline 0 & 0 \end{array}\right) \tag{5.25}$$

In the DF implementation, D is as an accumulator for the convolution. To even further simplify and optimize, we run Equation 5.23 iteratively $\max s$ times, and apply Equation 5.24 only once after the last iteration of the convolution. Assuming that input data has zero-mean, this does not impact numerical stability. On the other hand, care must be taken and a thorough numerical stability analysis should be conducted.

5.2.5.3 Results

These different variants were benchmarked in order to give insights about the cost of the DF operation. This information is used in order to determine the maximum number of hypotheses that can be used in real-time at this stage. Figure 5.14 gives the evolution of relative execution time specifically for the Doppler Filtering operation.

We notice on this figure that the cuDNN and CUTLASS versions exhibit similar performance. However, the CUTLASS version applies the Sparse DF operation, which can provide great reduction of the effective filter size, and proportional reduction of the computation time.

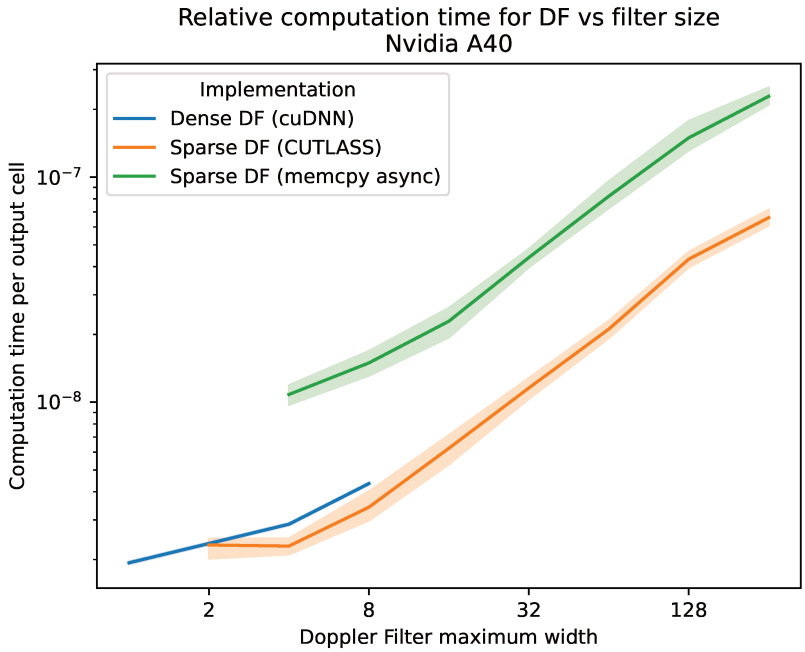


Figure 5.14: Evolution of the relative execution time for DF as of the maximum DF filter width. Note that both dense DF and sparse DF operations are compared. In the case of sparse DF, the width corresponds to the compressed filter width $\max(s)$.

5.2.6 LogMod

The LogMod operation in itself is not compute intensive at all, as it is just an embarrassingly parallel operation. However, it is rather memory intensive, as each cell processed requires a read of 8 B, and a write of 4 B. This is a strongly memory-bound kernel. As such, it is a very strong candidate for kernel fusion, as explained in Section 5.2.9.

In cases where kernel fusion cannot be implemented, we provide an optimized implementation, using optionally vectorized memory accesses. Input complex samples can be read by pairs, resulting in 16 B memory reads, which is both the maximum and the optimal configuration on current Nvidia GPUs.

After the LogMod operation, the pair of floats are written back, resulting in 8 B memory writes, which is also a good configuration.

5.2.7 CFARs

Two variants of CFARs were implemented: CA-CFAR and OS-CFAR.

5.2.7.1 CA-CFAR

CA-CFAR can be viewed as a 1D convolution with a specific filter. As such, it could be implemented in a straightforward way using any of the iterations presented in Section 5.2.3.1.

The filter could also be recomputed in-place in order to avoid the memory reads associated to reading the filter, and thus obtain better performance. However, this was not implemented, as we relied on our optimized convolution kernel.

5.2.7.2 OS-CFAR

OS-CFAR, being based on rank statistics, are harder to implement on GPU. An internship at Thales focused on porting a sequential (per signal row) histogram-based OS-CFAR from CPU to GPU. We also provide a quickselect implementation, working independently for each cell of the

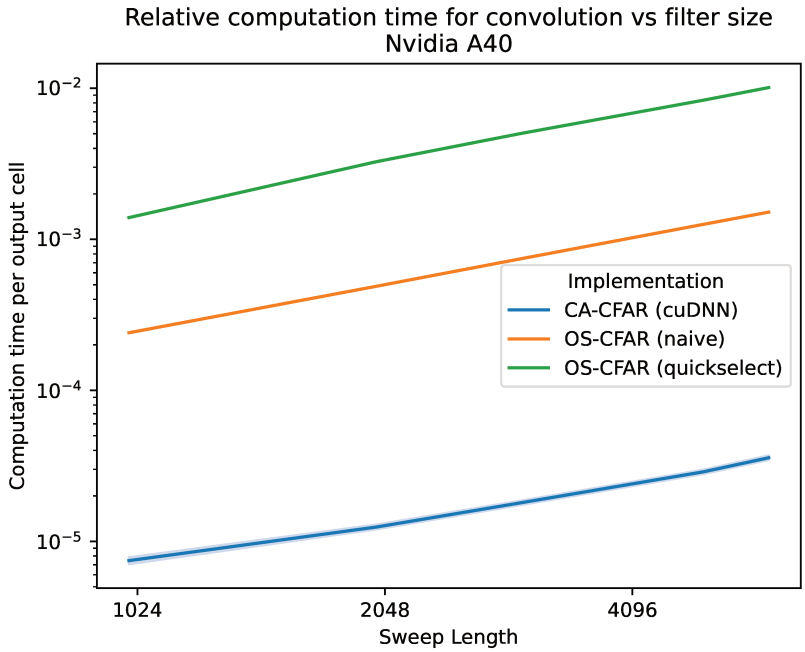


Figure 5.15: Relative execution time of different CFARs as of signal length

signal. Finally, research towards sorting networks and new algorithms for windowed rank statistics algorithms has started.

5.2.7.3 Results

Again, the different implementations of CFARs were benchmarked, and the relative execution time is reported on Figure 5.15.

It can be noticed that the CA-CFAR is more than order of magnitude faster than any of the OS-CFAR. This makes the latter more constraining to use, even if it provides more interesting features. The computation time for these implementations evolves linearly.

5.2.8 Full pipeline

In order to give a better intuition on the importance of each block of the pipeline in terms of contribution to the total computation time, we give a bar chart representing the percentage of execution time dedicated to each block. This bar chart is given in Figure 5.16.

On this figure, and this specific configuration, we notice that the computation time is largely dominated by the PC, and followed closely by DF. Note that both blocks presented here use the cuDNN version, and that faster implementations have been developed since.

It can also be noted that the LogMod operation, although it is very simple, takes up a significant portion of the pipeline's execution time. This motivated the use of a very interesting GPU kernel optimization technique: kernel fusion.

5.2.9 Kernel fusion

As shown through this section, some building blocks show significant footprint in the overall pipeline execution profile even though their complexity is very limited. This is especially the case for LogMod and detection, and to a lesser extent for CFARs.

A major optimization effort was to set up kernel fusion, in order to reduce the number of global memory accesses, and help reducing the cost of these computationally-low, memory-bound kernels.

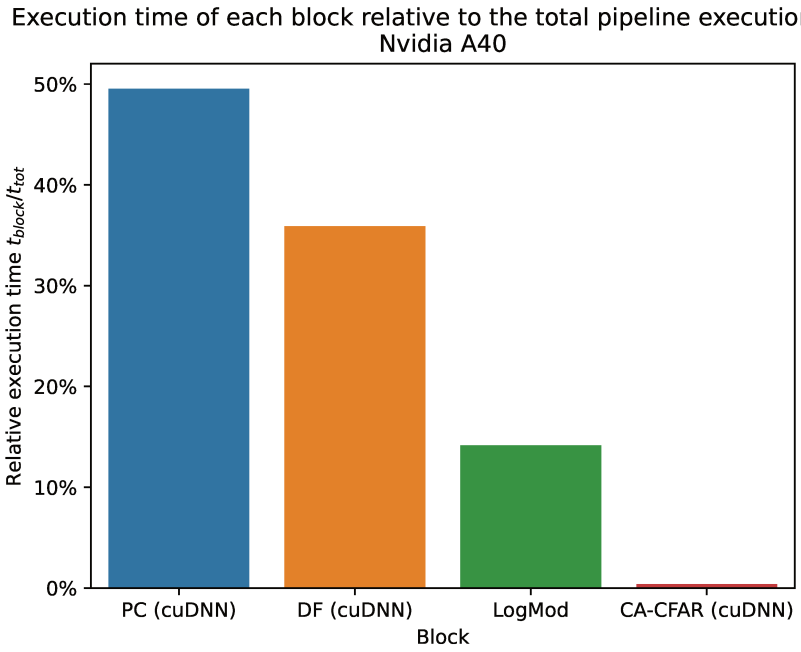


Figure 5.16: Execution time of each block relative to total execution time on a specific configuration. This configuration is chosen as an average case in the parameter grid considered during this work. $N_{streams} = 64$, $N_{PC} = 5$, $N_{HDF} = 5$, $N_{DF} = 128$, $\max(s) = 11$, $N_{Sweep} = 40$, SweepLength = 5000.

5.2.9.1 DF + LogMod

An ideal combination on which to perform kernel fusion is to add the LogMod operation after computing the DF. Indeed, DF outputs complex samples, while LogMod applies to these samples and reduce them to real samples.

Considering an example with N output samples from the DF stage, we can derive the number of bytes loaded with and without kernel fusion:

$$\begin{aligned} W_1 &= (8N + 8N + 4N)\,\mathrm{B} \\ &= 20N\,\mathrm{B} \\ W_2 &= 4N\,\mathrm{B} \end{aligned}$$

$16N$ B worth of memory transfers can be avoided using kernel fusion between these two building blocks. This can easily represent multiple gigabytes of avoided data transfer, which directly improves performance. The fused DF + LogMod kernel is actually faster than the single DF, as only half the amount of bytes is written to the GPU global memory.

5.2.9.2 CFARs + Detection

In a similar way, the multiple CFARs each produce one sample per range cell, each consumed by the detection stage, where a simple threshold is applied. This makes it a good candidate for kernel fusion.

Theoretically, following the same line of reasoning than in the previous section, considering a case with M CFARs and N output samples from each CFAR, the transfer of $8MN$ B from the GPU's DRAM can be avoided.

5.2.9.3 Other

Other fusions were tried, towards a goal of a single GPU kernel for the whole SP. But the relatively complex memory accesses related to convolution made these efforts unsuccessful, commonly because of a two great mismatch between access patterns.

5.2.10 Future work

This sums up the efforts deployed for implementation of a Massively Multi-Hypotheses Radar SP. In this final section about this work, we list the remaining ideas and directions worth to be explored in order to further improve it.

5.2.10.1 Reduced precision

During this work, the most common data type used was the classic IEEE 754 32-bit floating point. Indeed, radar systems, in addition to requiring real-time execution, must be accurate enough. Strong efforts have been deployed to study numerical stability of FP32-based implementations[50], especially for convolutions in the field of AI[63], where reduced precision is applicable, or even fixed-point arithmetic[37].

In this work, we propose one implementation based on the TF32, which already is a reduced precision floating point value, in order to target Tensor Cores, but numerical stability.

A full numerical stability study covering all alternative floating point representations available in CUDA (FP16, BF16, and TF32) would be of great interest: if not affecting results too much, these could provide a 2x speedup if considering regular CUDA cores, and up to 8x if considering Tensor Cores, compared to the same CUDA core based FP32 function.

5.2.10.2 CUTLASS + CuTe

A new major version of CUTLASS was released as this PhD project was reaching completion, CUTLASS 3.0. It shipped with a new sister library, CuTe. This library has the goal to unify the many complicated data arrangements presented in the last section (paragraph 5.2.5.2.4), previously handled by classes sharing a same legacy `C++` concept, hardcoded for each different case.

CuTe's new `Tensor` class can be modified to implement the $\widehat{L}$ operator of Equation 5.21 efficiently and seamlessly. It may even be possible to use CUTLASS' device-level API with cleverly arranged tensors to directly implement the DF using CUTLASS' high performance GEMM implementation, and get closer to the maximum throughput of the GPU. Moreover, this would enable easy switching from one data type to another, which was mentioned in the previous point.

Finally, this would also allow the developer to automatically benefit from new additions to CUTLASS, and as such automatic support and optimizations for new GPU architectures. However, this involves some kind of hacking around the actual interface of CUTLASS, and care would have to be taken on GPU or library upgrade. Our modifications may not be compatible with newer hardware features such as Tensor Memory Accelerator (TMA).

This has not been done yet because the library was released at a late stage of this work, and also because support for GPUs older than the Hopper architecture (CC <9.0) were not supported by the new API featuring CuTe. However, this should come in the next set of releases and could bring great performance improvement.

5.2.10.3 Overlap-save with cuFFT

In order overcome the limited feature set of cuFFTDx and to support wider filters, it would be beneficial to provide and implementation of the overlap-save method based on cuFFT. This would follow exact the same principle, and the overhead of using cuFFT instead of cuFFTDx should be less important for larger FFT, as more global memory accesses would be required because of the bigger memory footprint in both cases, effectively reducing the impact of using cuFFTDx.

5.2.10.4 Unit testing

Even if all implementations were checked for correctness, the quantity of different implementations made it difficult to keep consistent code quality. One of the toughest hurdle for this work to be used on critical applications such as radar systems is their testing. Not all of these implementations have been integrated in an automated testing solution, making debugging and new feature addition more difficult.

A good candidate for such an automated testing solution is CTest, a utility shipped with CMake, extensively used during this project. Using CTest provides enough features in order to provide automated testing, and does not require any additional dependency.

Chapter 6

FRB detection on NenuFAR

In this chapter, we reproduce a journal paper submitted to the Journal of Astronomical Telescopes, Instruments and Systems (JATIS). This article was the final achievement of a major part of this thesis, the detection of FRBs on the NenuFAR telescope. It relies on methods presented in other parts of this document, including the high-performance data acquisition system, classical CUDA kernel performance optimization and random-sampled benchmarking.

The main contributions of this paper are:

- A means to detect more FRBs, and help gaining a better understanding of these signals
- A solution to do so at low observational frequency, on NenuFAR, which poses important challenges because of physical effects (dispersion, scattering)
- A method to compute incoherent dedispersion on very long time scales while keeping a reduced memory footprint
- The development of highly optimized GPU kernels for FRB detection
- The integration of our data acquisition system in a real system

Additionally, we detect three known pulsars, used to validate the functional behavior of the pipeline. Indeed, pulsars share a similar signature with FRBs, with the advantage of being periodic, improving reproducibility.

24/7 Fast Radio Bursts search on NenuFAR

Julien Plante[a], Damien Gratadour[a], Lionel Matias[b], Philippe Zarka[a,c], Cédric Dumez-Viou[c], Louis Bondonneau[c], Jean-Mathias Griessmeier[c,e], Elena Agostini[d], Baptiste Cecconi[a,c], Julien N. Girard[a], Stéphane Corbel[c,f], Vyacheslav Zakharenko[g], Oleg Ulyanov[g], Oleksandr Konovalenko[g]

[a]LESIA, Obervatoire de Paris, Université PSL, CNRS, Sorbonne Université, Université Paris Cité, Meudon, Hauts-de-Seine, France
[b]Thales LAS France, Limours, Essonne, France
[c]ORN, Observatoire de Paris, Université PSL, CNRS, Université d'Orléans, Nançay, Cher, France
[d]NVIDIA, Santa Clara, CA, USA
[e]LPC2E, Université d'Orléans, CNRS, Orléans, Loiret, France
[f]AIM, CEA, CNRS, Université Paris Cité, Université Paris-Saclay, F-91191 Gif-sur-Yvette, France
[g]IRA NASU, Institute of Radio Astronomy of National Academy of Sciences of Ukraine, Mystetstv 4, 61022 Kharkiv, Ukraine

Abstract. We present a real-time Fast Radio Burst (FRB) detection pipeline designed for NenuFAR, that performs a continuous blind search for these elusive radio sources. We achieve a rough detection, with the goal to store the raw time series selectively, in order to only record signal segments containing potential FRBs, with the goal to perform a refined detection and study later, offline. This greatly reduces the amount of storage needed for FRB surveys in the low-frequency radio domain, and will help astronomers to focus on the study of these signals, and not on their detection.

We propose a variant for the parametric dedispersion algorithm, one of the main bottleneck of FRB detection pipelines, to make it more usable on highly dispersed signals ($\Delta t > 1\,\mathrm{min}$), a common case in low-frequency observations.

To implement this pipeline, we leverage High-Performance Computing methods such as DPDK, GPUDirect RDMA, GPUDirect Storage and CUDA, that made it possible to reach real-time performance on this specific experiment. We provide a public repository for the source code of this pipeline, containing reusable implementations based on the CuTe library.

We present a detailed performance analysis of the different components of our pipeline, and show how our implementations could scale to more intense workloads such as in the context of SKA.

Finally, we deployed this pipeline on NenuFAR, and report the results of validating this work by detecting known pulsars in real-time, which share similar features with FRBs, albeit with generally smaller DM. We measure each pulsar period and DM, and report less than 1 % relative error compared to values from PSRCAT.

Keywords: data processing, astronomy, parallel processing, arrays, data recording, local area networks.

1 Introduction

Fast Radio Bursts (FRB) are a kind of radio transient events characterized by short, powerful and highly dispersed emissions over a broad spectrum. The corresponding emission mechanism is not identified yet, hence detecting new FRBs and collecting more data, especially in yet unexplored frequency ranges, is a major challenge in modern radioastronomy[1].

Since the discovery of the first FRB in 2007, several hundreds of FRBs have been detected,[2] but most of them were detected in a frequency band of the order of 0.5–1 GHz, with the current lowest detection being on LOFAR in the range 110–188 MHz[3]. One of the scientific goals of NenuFAR is to search for FRBs at much lower frequencies (10–85 MHz), but until now, no FRB has been detected throughout the observation campaigns, which are still being analyzed. This search has started in 2019, with a telescope time budget of a little less than a thousand hours per year.

1.1 Previous work

The current FRB search methodology on NenuFAR is split in two stages. First, the UnDySPutEd[4] system reduces NenuFAR digital, beamformed, complex waveforms recorded by the LaNewBa[5,6] backend to spectra of requested time and frequency resolution, in real-time. Spectra are stored on disk, and processed offline with a set of IDL scripts[7] or Python scripts.[6,8]

The UnDySPutEd system is composed of two servers, with two Nvidia GeForce GTX 1080 GPUs each. Data acquisition is done on four separate ports to cope with the maximum bandwidth of 4×2.2 Gbit/s. On the one hand, the Linux networking stack is used with specific tuning, and some packet losses still happen on rare occasions. On the other hand, the GPU computing part processes data, and achieves real-time performance successfully.

Regarding the acquisition part of this system, many approaches have been developed to improve performance of the acquisition of UDP packets, from advanced Linux kernel tuning,[9] userland networking[10–12] to domain specific hardware.[13]

We can also cite the Remote Direct Memory Access (RDMA) mechanism, enabled by some standards, the most popular being Infiniband, RDMA over Converged Ethernet (RoCE) and iWARP. However, RDMA-based methods require a modification of the LaNewBa beamformer (see Figure 1), the emitter of our data products. Such a modification is out of the scope of this work.

Userland networking has been successfully used in a few radioastronomical projects, such as for the correlators of CHIME[14] and MeerKAT,[15] or the pulsar reduction system of FAST.[16] However in these projects, data is always transferred to CPU memory before being copied to the GPU memory (when GPU computing is used), which implies increased memory footprint and latency, and can impact scalability.

Using custom FPGAs, enabling DMA to the GPU memory has been implemented[13] with great results, at the cost of increased complexity, low level of maintainability, and costly development cycle.

Regarding the computing part, many frameworks and pipelines were developed for single pulse detection,[17] that could replace the offline processing stage. But most of these are designed for a specific telescope, which makes them harder to reuse, especially on NenuFAR where dispersive delays can be very large. Indeed, NenuFAR observes a lower frequency domain (10–85 MHz) while most current radio telescopes traditionally used for FRB science observe the sky in a range close to the gigahertz. The impact of this peculiarity of NenuFAR is shown in Table 1.

Telescope	Frequency range	Pipeline	Max. Disp. Delay Δt $DM = 1000\,\mathrm{pc\,cm^{-3}}$
Arecibo	0.3–10 GHz	PRESTO†[18]	46.0 s
ASKAP	0.7–1.8 GHz	FREDDA[19]	7.2 s
CHIME	400–810 MHz	BONSAI[14]	19.6 s
FAST	70–3000 GHz		14 min 6 s
GBT	0.29–115 GHz	PRESTO† / GREENBURST[20]	49.3 s
LOFAR HBA	110–188 MHz	PRESTO + DEDISP†[3]	3 min 45 s
MeerKAT	1–10 GHz	MEERTRAP[21]	4.1 s
NenuFAR	**10–85 MHz****	**this work**	**11 h 22 min**
Parkes UWL	0.7–4 GHz	HEIMDALL[22]	8.2 s
SKA	50–350 MHz		27 min 5 s
UTMOST	828–858 MHz	HEIMDALL[23]	400 ms
UTR-2/URAN-2	8–33 MHz	data analysis†[24]	16 h 57 min
Westerbork	0.12–8.3 GHz*	AMBER[25]	3 min 36 s

Table 1: List of radio telescopes, corresponding FRB detection pipelines, and associated maximum dispersion delays. Considering the whole frequency band, dispersion delays on NenuFAR are two orders of magnitude longer than on the Westerbork Telescope, the next telescope in operation sorted by maximum dispersion delay. The dispersion delay is proportional to $\frac{1}{f_{min}^2} - \frac{1}{f_{max}^2}$, as developed in (1).

*Westerbork Telescope has an instantaneous bandwidth of 120 MHz inside this range.

** In practice, the frequency range is often cropped to 40–85 MHz, for cleaner data and a maximum dispersion delay reduced to 34 min at $DM = 1000\,\mathrm{pc\,cm^{-3}}$, which is still one order of magnitude over LOFAR.

† Search performed on recordings.

1.2 Contributions

In this work, we demonstrate the use of an emerging technology, GPUDirect RDMA from a Network Interface Controller (NIC) to a GPU and its recent support in DPDK,[11,26] in the context of data acquisition for radio-astronomy data and FRB detection. Using this technology, we receive NenuFAR's Beamformed High Resolution (BHR) data using a single 10 Gbit/s port, with no packet loss, DMA to the GPU and minimal CPU usage. BHR data is a stream produced by the LaNewBa beamformer, and consists of a set of waveforms along a configurable frequency band. This is the lowest-level data product available to users

of NenuFAR, before higher-level data streams such as visibilites produced by the Nickel correlator.

This data acquisition approach can be reused as-is for other observations on NenuFAR (exoplanet detection, sun observation), and can be adapted to any other applications requiring the acquisition of huge data streams. The compute part of the pipeline can be adapted for general spikes detection of potentially dispersed signals, which encompasses observations of the Sun, Jupiter, or atmospheric showers. To facilitate the adoption of this technology, we provide our code under LGPL (see section 5). Outside of the scope of this publication, we applied successfully this same method to Adaptive Optics with a focus on low latency, and radar systems with a focus on high bandwidth.

The main contribution of this work is the deployment of a real-time, low-frequency FRB detection pipeline for NenuFAR. This real-time pipeline is able to piggy-back on the observation time of agreeing experiments. By increasing the observation time, we increase the probability of FRB detection by the same factor. We target an uptime of 100 % for our real-time detection pipeline. However, some emitter, such as the Sun or Jupiter, are too intense to detect FRBs at the same time, but others, like exoplanet surveys, could be used, totalling for about 50 % of the observation time of the telescope. This is 5 times more than the current value, limited by an offline processing scheme.

Moreover, we propose a solution to perform incoherent dedispersion on signals featuring very long dispersive delays (>1 min), based on an overlap-add method applied to Dispersion Measure Transform (DMT). This removes the need for regular overlap, and helps reducing the memory footprint of the pipeline, which is one of the main limiting factor when working with huge dispersive delays.

As a minor contribution, we demonstrate the use of GPUDirect Storage (GDS), a technology used to perform DMA between a GPU and storage. We rely on this to provide observability of the pipeline while ensuring real-time performance.

2 Pipeline design

2.1 Hardware

A server has been installed on-site at Nançay to execute the pipeline described in this document. The configuration of this server is summarized in Table 2.

Type	Model	Quantity	Notes
CPU	AMD EPYC 7763 64-Core Processor	2	
GPU	NVIDIA A100 80GB PCIe	2	
NIC	MT42822 BlueField-2 integrated ConnectX-6 Dx	1	2×25 Gbit/s
DISK	NVMe SSD Controller PM173X	1	

Table 2: Server configuration: Thetis

Figure 1 gives an overview of our network setup. The NenuFAR array consists of about 1900 antennas, grouped in subsets of 19 antennas. The signal coming from these antennas is beamformed using an array of FPGAs on LaNewBa. This is able to output 1 to 4 beams, pointed in a specific direction. These beams are transferred over 4 network streams, called Beamformed High Resolution (BHR). Each BHR stream carries 192 frequency channels, for a maximum of 768 frequency channels split over the different beams. Each BHR stream represents about 2.5 Gbit/s, for a total of about 10 Gbit/s. These BHR streams are accessible multicasted through a specific network switch.

One of the ports of our NIC is connected through a 10 Gbit/s link to this switch. By subscribing to these streams using the IGMP protocol, our dedicated machine "Thetis" can receive the beamformed waveforms.The IGMP protocol is used to control multicasting over a network.

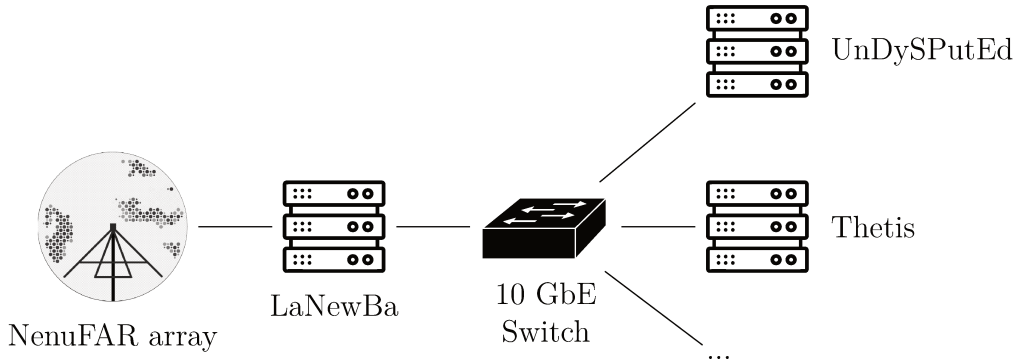


Figure 1: Network configuration overview. LaNewBa is NenuFAR's beamformer, an array of FPGAs.

Regarding the PCIe topology inside Thetis, the two Nvidia GPUs are on the same PCIe Host Bridge than the NIC, and are connected through PCIe bridges. Seven bridges are crossed to join one of the NIC's port to one of the GPUs. Figure 2 summarizes the PCIe topology of the machine Thetis.

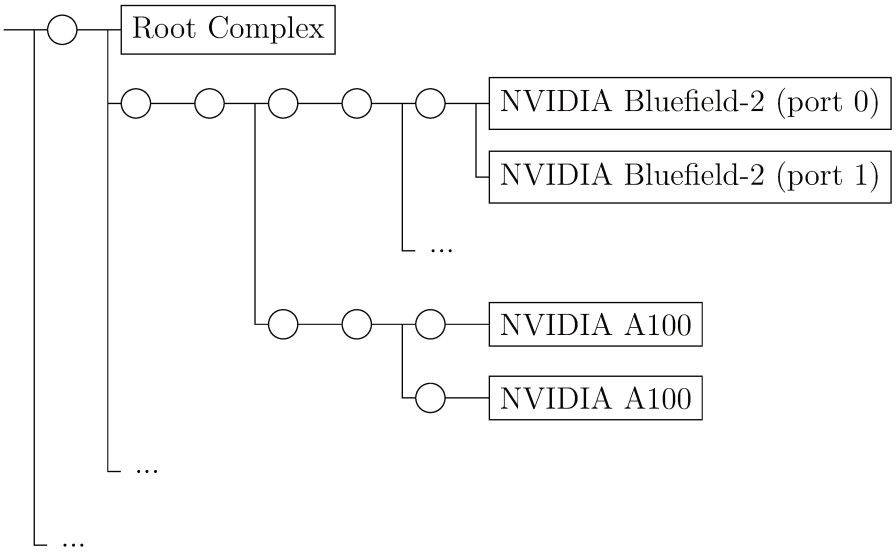


Figure 2: Thetis' PCIe topology summary. The symbol ◯ denotes PCIe bridges. Generally speaking, a communication between two PCIe devices will have better performance when less PCIe bridges are involved, as this reduces the risk of interference with other transactions. Moreover, a communication to a part of the PCIe topology with a different Root Complex (leftmost branch) will typically be slowed down by an order of magnitude, as a different type of interconnect is used (QPI/UPI).

2.2 Real-time data acquisition

In this section, we describe the real-time data acquisition system of the pipeline.

NenuFAR's beamformer relies on standard IP multicasting to send data to all its receivers. During initialization, our data acquisition system subscribes to this stream of data using the IGMP protocol. Then, during the entire execution of the application, network packets from this stream are continuously received, parsed, and the payload is stored in a ring buffer. Alongside the data ring buffer is a metadata ring buffer, containing relevant information about the data being received, such as the timestamp or the datatype used for the transfer.

Using DPDK and GPUDirect through the use of the DPDK library gpudev,[27] whole packets are directly transferred to the GPU memory.[28] The packet processing is then performed directly by the GPU, using a persistent CUDA kernel. We chose to use a persistent kernel to avoid the overhead of launching a CUDA kernel at high frequency.

The packet processing kernel follows these steps:

- Header parsing
- Data type conversion: one sample from NenuFAR is composed of four values: one complex number for each polarization. Complex numbers are stored as pairs of integers, of variable size (4, 8 or 16 bit) from one observation to another. Every sample is converted to a floating point type. Currently, only FP32 has been tested, but FP16 and FP64 implementations are also available.
- Output to a second ring buffer, used to store the raw spectrograms in a compact fashion, stripped from headers.

2.3 *Real-time processing*

We implemented a classical pipeline for FRB detection,[17] albeit with some variations to the different stages compared to state of the art. A high-level description is given in Figure 3, and an execution timeline is presented in Figure 4.

The main symbols used in this section are regrouped here:

- f_{min} and f_{max} are respectively the minimum and maximum frequencies observed. N_f is the number of frequency channels. Δf is the frequency resolution, and $\frac{f_{max}-f_{min}}{N_f} = \Delta f$.
- t is used to represent a time or duration, and N_t is a number of time samples corresponding to a certain duration. Δt is the time resolution, and $t = N_t \Delta t$.

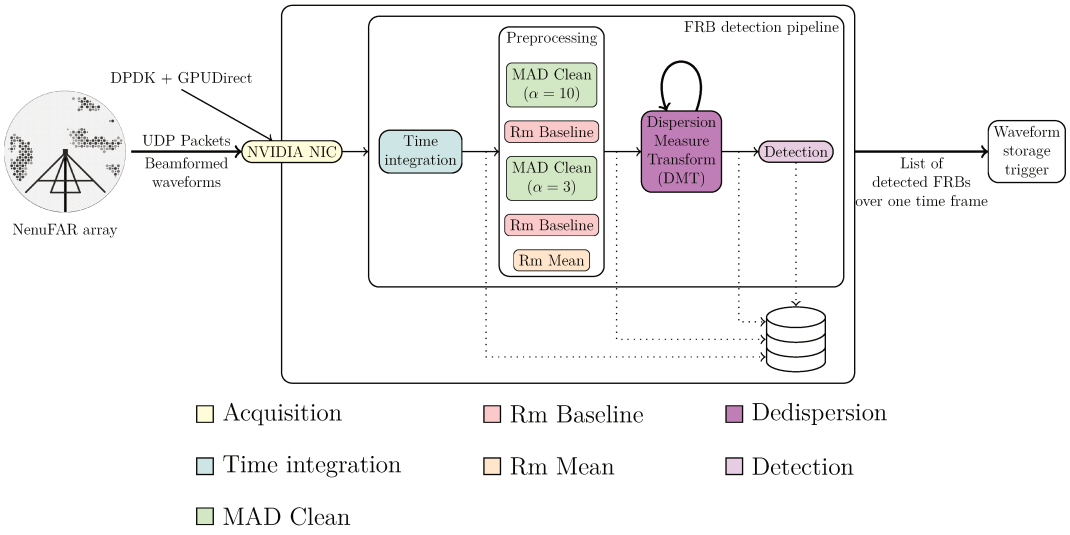


Figure 3: FRB detection pipeline overview. All components are described section 2.3. The dotted lines represent an optional dump to disk described section 2.3.4.

2.3.1 *Preprocessing*

Time integration Most radio transients are events featuring a duration in the range 1–100 ms. However, the native time resolution of NenuFAR is $\tau = \dfrac{1024}{200 \times 10^6} = 5.12 \times 10^{-6}$ s, which is three orders of magnitude shorter than the phenomenon we are trying to observe. The intra-channel dispersion further increase the number of time samples containing a pulse. As an example, at 40 MHz and a DM of 100 pc cm^{-3}, the intra-channel dispersion delay is of 3.90 ms with the native frequency resolution of NenuFAR, $\Delta f = \frac{1}{\tau} = 195.3125$ kHz. Trying to observe at native resolution leads to having a potential transients' signal below the noise level.

Two solutions were implemented to reduce time resolution for this pipeline:

- Taking the mean over successive chunks of data of size N_{fold} along the time axis. This directly reduce the time resolution by a factor N_{fold}, but also the data rate and memory usage.

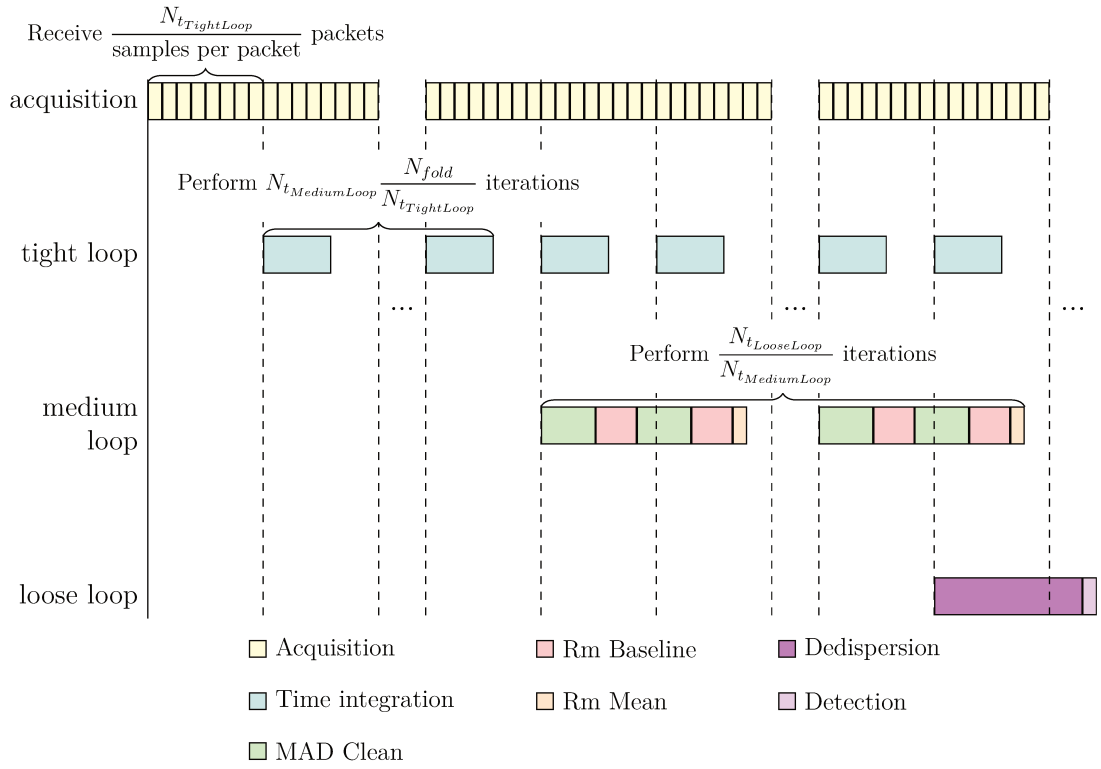


Figure 4: Execution timeline. Tasks are listed in descending priority. Priorities are enforced using CUDA streams priorities: operations are enqueued in CUDA streams, which are similar to FIFOs. Streams with the highest priority are dequeued first. However, preemption is possible but not guaranteed. Because of this, we use ring buffers to store incoming data, and must find a tradeoff between memory footprint and correctness. This tradeoff is tuned through the parameters $N_{t_{<...>Loop}}$, which enable selection of the number of time samples processed at once in each step, and a redundancy parameter, describing the number of repetitions of $N_{t_{<...>Loop}}$ each ring buffer can contain.

- Applying a Short-Time Fourier Transform (STFT) to the same chunks of data. This reduces the time resolution by a factor N_{fold}, but at the same time increases frequency resolution by the same factor. Increasing the frequency resolution is very valuable, as it reduces the width of the frequency channels and, consequently, the intra-channel dispersion. No information is lost, thus bandwidth and memory usage does not change.

In practice, only the first method has been used, because of artifacts arising using the STFT. Windowing and/or overlap will have to be implemented to make this method viable. Once corrected, a combination of both methods may help finding the good compromise between signal resolution and data reduction.

The choice of N_{fold} is critical, given the relatively large set of signal duration we need to detect.

This block takes a tensor of shape (N_f, N_t), and outputs a tensor of shape ($N_f \times N_{fold_{STFT}}$, $\frac{N_t}{N_{fold}}$). When used in this pipeline, N_t is chosen as $N_{t_{TightLoop}}$, introduced in Figure 4.

Radio Frequency Interference mitigation (MAD Clean) NenuFAR, as most radio-telescopes (but even more so at very low frequencies), is subject to Radio Frequency Interference (RFI). These intense, characteristic signals can significantly impact single pulse detection, as their intensity is orders of magnitude larger than that of FRBs.

To mitigate effects of RFI, we use a simple clipping approach similar to that of PRESTO's `rfifind`.[29] The median and Median Absolute Deviation (MAD) are computed over I, the intensity of the spectrogram. Samples outside the range [median $\pm$ αMAD] are identified as outliers, and action is taken to remove them from the signal. The parameter α gives the range of the confidence interval used to discriminate inliers from outliers. Assuming normally-distributed noise, it is possible to derive the confidence level associated to the interval,[30] helping the choice of α: $\mathrm{CL} = \mathrm{erf}\left(\alpha \cdot \mathrm{erf}^{-1}(1/2)\right)$, where erf is the error function.

Multiple methods are available to remove RFIs. The original method was a clipping, but was found to have a neBative impact on the third stage of the pipeline, the Dispersion Measure Transform (DMT), introduced on Figure 3. Even after this clipping, RFIs dominate the DMT, which makes detection more difficult. The most accurate implemented mode is to replace potential RFIs with NaNs, to flag them and instruct the next stages to avoid using this part of the signal. However, this complicates downstream implementations on GPU, and thus has not been implemented. The chosen method was to replace potential RFIs with the median of neighboring signal. This has minimal effect on the statistics used during the pipeline and on the implementation complexity. A visual comparison of these methods is given in Figure 5.

More advanced methods have been developed for RFI mitigation, both relying on regular algorithms[31–35] and machine learning.[36–38] These methods would be interesting to implement in future iterations of this pipeline to further increase SNR of the detected single pulses.

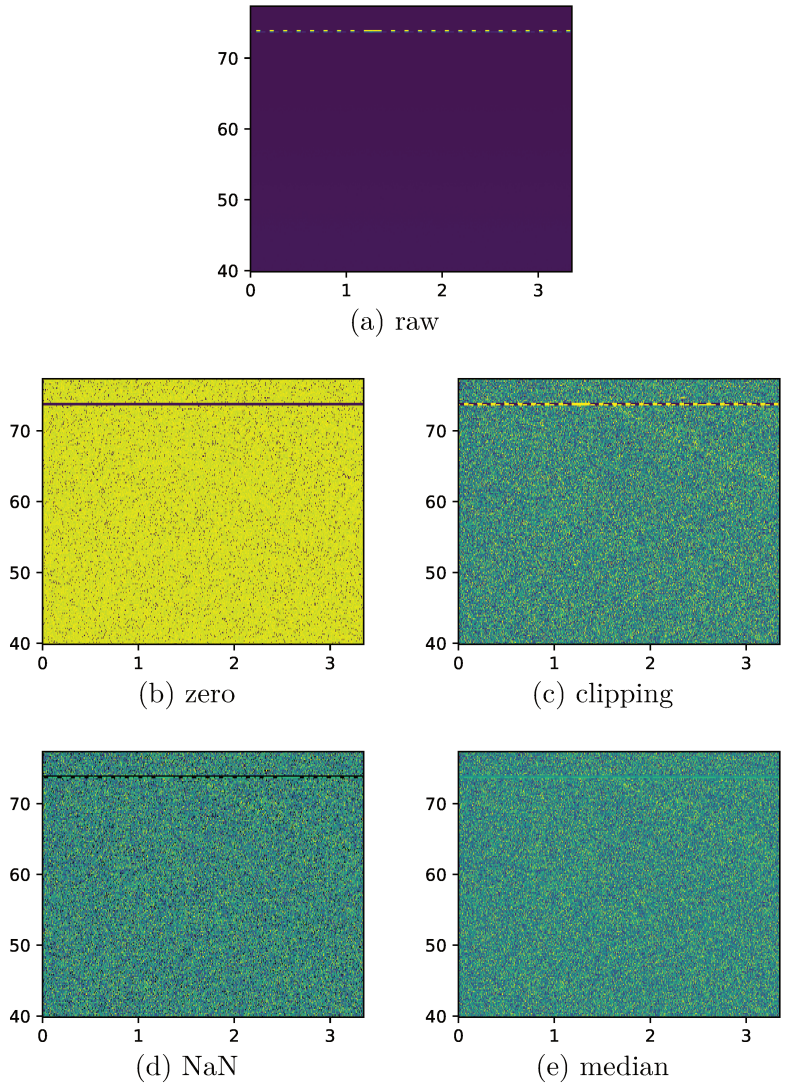
This block takes a tensor of shape (N_f, N_t), and outputs a tensor of the same shape. When used in this pipeline, N_t is chosen as $N_{t_{MediumLoop}}$, introduced in Figure 4.

Frequency gain (Rm Baseline) A second effect has to be addressed: NenuFAR does not have the same gain at all frequencies. The intensity in each channel is corrected by the following sequence of operations:

- Compute the mean of each frequency channel
- Compute the global median
- Subtract each sample by the mean of its frequency channel
- Scale by the ratio between the aforementioned mean and the median

These two operations are repeated with decreasing values of α to incrementally improve the signal-to-noise ratio.

This block takes a tensor of shape (N_f, N_t), and outputs a tensor of the same shape. When used in this pipeline, N_t is chosen as $N_{t_{MediumLoop}}$, introduced in Figure 4.



(a) raw

(b) zero

(c) clipping

(d) NaN

(e) median

Figure 5: Result of a preprocessing chain applied on with varying RFI mitigation methods. The input signal features strong, pulsed RFIs around 74 MHz, and a dispersed signal starting at $t_0 = 1.2\,\mathrm{s}$ and $DM = 5.9\,\mathrm{pc\,cm^{-3}}$. On the horizontal axis is the time in seconds, and the frequency in megahertz on the vertical axis.
It can be seen on the output of the clipping method that the RFIs still dominate the signal. This has a strongly negative impact during the dedispersion. Approaches based on replacing values by NaNs or the median of the signal exhibit the expected dispersed signal, while mitigating the RFIs.

Zero centering (Rm Mean) The last preprocessing stage is to remove the global mean of the signal, in order to center data around 0. Because the DMT is a sum, centering data around 0 helps improving numerical stability by summing smaller values. This operation is valid at least for the DMT algorithm we used, described in the next section, especially through equation (4):

$$DMT\left\{I - \bar{I}\right\} = DMT\{I\} - N_f \cdot \bar{I}$$

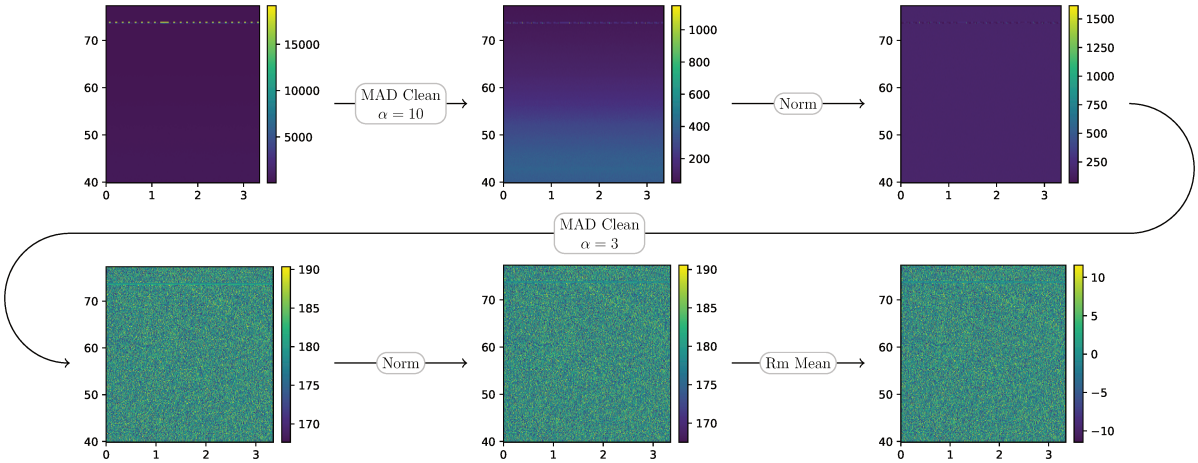


Figure 6: Full preprocessing chain. The input is the time integrated Stokes I spectrogram. The output is RFI mitigated, normalized, and centered around 0.

Where $\bar{I}$ is the arithmetic mean of I.

This block takes a tensor of shape $(N_f,\ N_t)$, and outputs a tensor of the same shape. When used in this pipeline, N_t is chosen as $N_{t_{MediumLoop}}$, introduced in Figure 4.

The combination of these operations is an attempt at cleaning the input signal, and helps increasing SNR to make single pulses detectable. The effect of these operations is shown graphically on Figure 6

2.3.2 Streaming DMT

We implemented a variation of the naive Dispersion Measure Transform (DMT).[39] This variation adresses three shortcomings of the standard DMT: the need for overlap, the necessity of storing the whole buffer on which we compute the DMT before starting the computation, and the selection of only one time sample per frequency channel.

Because of the extremely long dispersion delays observed on NenuFAR, buffers of several gigabytes are required to store the intermediate arrays. The first two shortcomings listed above make the use of the naive DMT very impractical. We modified the algorithm to be able to compute the DMT iteratively over time, processing smaller chunks of data. Finally, the last shortcoming appears at higher DM. With a small number of frequency channels, the intra-channel dispersion is not taken into account, but this type of dispersion can reach up to a minute on NenuFAR. There is a need to enable the selection of multiple time samples per frequency channel, that we address with our variation.

The Dispersion Measure Transform stems from the idea of integrating along each possible dispersion curve, parametrized using two parameters: t_0, the arrival time of the highest frequency component, and DM, the dispersion measure.

Such a dispersion curve is characterized by the following equation:

$$D_{t_0,DM} : t - t_0 = \Delta(f, f_{max}, DM) \tag{1}$$

With $t \in \mathbb{R}$ denoting the time variable, $f \in [f_{min}, f_{max}]$ the observation frequency variable, $(f_{min}, f_{max}) \in \mathbb{R}^{+2}$ respectively the lowest and highest observation frequencies, $t_0 \in \mathbb{R}$ the

arrival time of the highest frequency parameter, and $DM \in \mathbb{R}^+$ the dispersion measure parameter. $\Delta(f_1, f_2, DM)$ is the dispersive delay between two frequencies f_1 and f_2, at dispersion measure DM, and is given by:

$$\Delta(f_1, f_2, DM) = kDM\left(\frac{1}{f_1^2} - \frac{1}{f_2^2}\right)$$

Where $k \approx 4149\,\mathrm{MHz}^2\,\mathrm{pc}^{-1}\,\mathrm{cm}^{-3}\,\mathrm{ms}$ is the dispersion constant.

The continuous Dispersion Measure Transform (DMT) is given by the following line integral:

$$A\left(t_0, DM\right) = \int_{D_{t_0,DM}} I(s)\mathrm{d}s \tag{2}$$

Where I is a continuous spectrogram of the sky.

In practice, the received data is discrete, and we reduce this expression to a discrete version. The usual parametrization chosen for $D_{t_0,DM}$ maps each frequency channel to a time sample:

$$\begin{cases} f_k = & f_{min} + k\dfrac{f_{max} - f_{min}}{N_f - 1} \quad \text{for } k \in 0, 1, ..., N_f - 1 \\ t_k = & t_0 + \Delta(f_k, f_{max}, DM) \end{cases} \tag{3}$$

This leads to the following expression of naive DMT:[39,40]

$$A(t_0, DM) = \sum_{k=0}^{N_f-1} I(t_k, f_k) \tag{4}$$

For the first iteration of this pipeline, we focused on the naive DMT algorithm, which has $\mathcal{O}(N_{DM}N_fN_t)$ complexity.[39] Newer algorithms feature reduced complexity, such as the FDMT,[40] with complexity $\mathcal{O}(N_{DM}N_t\log_2(N_f))$, or the FFT-FDMT,[40] with complexity $\mathcal{O}(N_tN_f\log_2(N_{DM}/N_f))$.

In practice, I must be fully available in memory to compute A. Moreover, overlap is needed to compute A for large t_0 without going out of bounds of I: to compute the DMT for $t_0 \in [0, t_{out}]$, samples of I must be available in memory in the range $[0, t_{out} + \Delta(f_{min}, f_{max}, DM_{max})]$. For high DM and low frequency, this becomes very impractical.

To address these first two shortcomings, we propose implementing DMT in the fashion of a running sum, and defining out-of-bounds accesses to I. This results in an algorithm that we call Streaming DMT in the rest of this paper, that computes iteratively the DMT A for $t_0 \in [0, t_{out}]$ with a smaller input signal window I, restricted to the time range $t \in [0, t_{in}]$, where $t_{in} \leq t_{out}$. As such, t_{in} and t_{out} respectively stand for the durations of the time blocks in input and output of our proposed dedispersion stage.

$$A_n(t_0, DM) = \begin{cases} A_{n-1}(t_0 + t_{in}, DM) + \sum_{k=0}^{N_f-1} \hat{I}_n(t_k, f_k) & \text{if } 0 \leq t_0 < t_{out} - t_{in} \\ \sum_{k=0}^{N_f-1} \hat{I}_n(t_k, f_k) & \text{if } t_{out} - t_{in} \leq t_0 < t_{out} \end{cases}, n > 0 \quad (5)$$

Under the following initial condition:

$$A_0(t_0, DM) = 0$$

$\hat{I}_n(t, f)$ is defined as:

$$\hat{I}_n(t, f) = \begin{cases} I_n(t, f) & \text{if } t < t_{in} \\ 0 & \text{else} \end{cases} \quad (6)$$

After a sufficient number of iterations $n = \frac{t_{out}}{t_{in}}$, and for $t < t_{in}$, $A_n(t, f)$ is the exact DMT, as if computed in one iteration with the necessary overlap. Figure 7 gives a visual representation of this algorithm.

Note that even if this algorithm was developed over the naive DMT, it is independent of the actual DMT algorithm used. The summation term in (5), corresponding to the DMT computation, could be replaced by other incoherent dedispersion algorithm, in order to benefit from a potentially reduced complexity.

To address the third shortcoming, we propose an alternative parametrization of the dispersion curve:

$$\begin{cases} f_k = & f_{min} + k\frac{f_{max} - f_{min}}{N - 1} \quad \text{for } k \in 0, 1, ..., N - 1 \\ t_k = & t_0 + \Delta(f_k, f_{max}, DM) \end{cases} \quad (7)$$

Where N is an arbitrary integer. Bigger values of N increase the resolution of the DMT and reduce aliasing, as shown graphically on Figure 8. Resolution is improved for $N > N_f$.

This block takes a tensor of shape (N_f, $N_{t_{in}}$), and outputs a tensor of shape (N_{dm}, $N_{t_{out}}$), where $N_{t_{out}} = kN_{t_{in}}$. When used in this pipeline, $N_{t_{in}}$ is chosen as $N_{t_{LooseLoop}}$, introduced in Figure 4, and k is dependant on the biggest DM trial value, and generally chosen as 4.

2.3.3 Detection

Detection is performed with a simple threshold over the Signal-to-Noise Ratio (SNR) values of each DMT sample. It is performed independently for each DM trial. We define the SNR by $SNR = \frac{x-\mu}{\sigma}$, where x is a sample of the DMT, and μ and σ are respectively the mean and standard deviation of the time series corresponding to one DM trial in the DMT output.

This approach has been chosen for the first iteration of this pipeline for its simplicity. Next iterations will have to use the usual combination of normalization + boxcar filtering.[17]

Candidate grouping is then applied in order to filter false positives, as single pulses generally appears in DMT bins close to the brightest bin.[17] We used a density-aware clustering

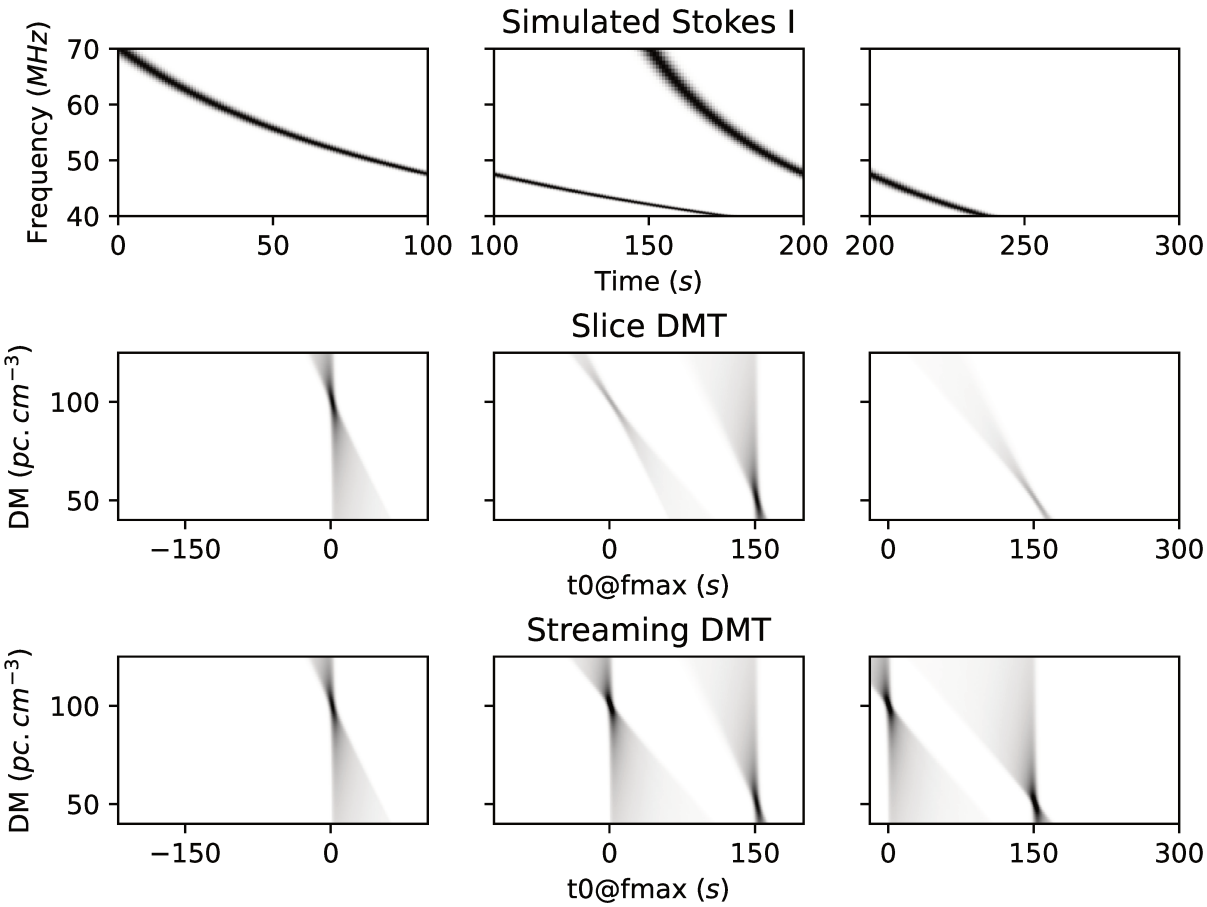


Figure 7: Streaming DMT applied on simulated input. Two FRBs are simulated, with parameters ($t_0 = 0\,\mathrm{s}$, $DM = 100\,\mathrm{pc\,cm^{-3}}$) and ($t_0 = 150\,\mathrm{s}$, $DM = 50\,\mathrm{pc\,cm^{-3}}$) respectively. To demonstrate the Streaming DMT, the simulated signal is split into three slices (top row). The DMT of each slice is computed (middle row), and shift accumulated (bottom row). Notice how the time scale is different between the input signal and the DMT: this enables covering dispersion curves where the maximum frequency component has been received in the past. By using a long enough time scale for DMT, we can reduce completely the need for overlap.

method, DBSCAN[41] to group close hits together, and flag outliers[42] directly from the FRB candidates extracted from the DMT output. By taking the centroid of these clusters, we achieve super-resolution and increase confidence on the detection. A probabilistic model such as gaussian mixture would also be interesting to consider, to search for the specific butterfly shape of the pulse in the DMT space, under the assumption that the chosen algorithm is also able to filter outliers. Deep learning methods have also been created for this task.[43,44]

This block takes a tensor of shape (N_{dm}, N_t) and outputs a vector of length $N_{candidates}$. When used in this pipeline, N_t is chosen as $N_{t_{LooseLoop}}$, introduced in Figure 4.

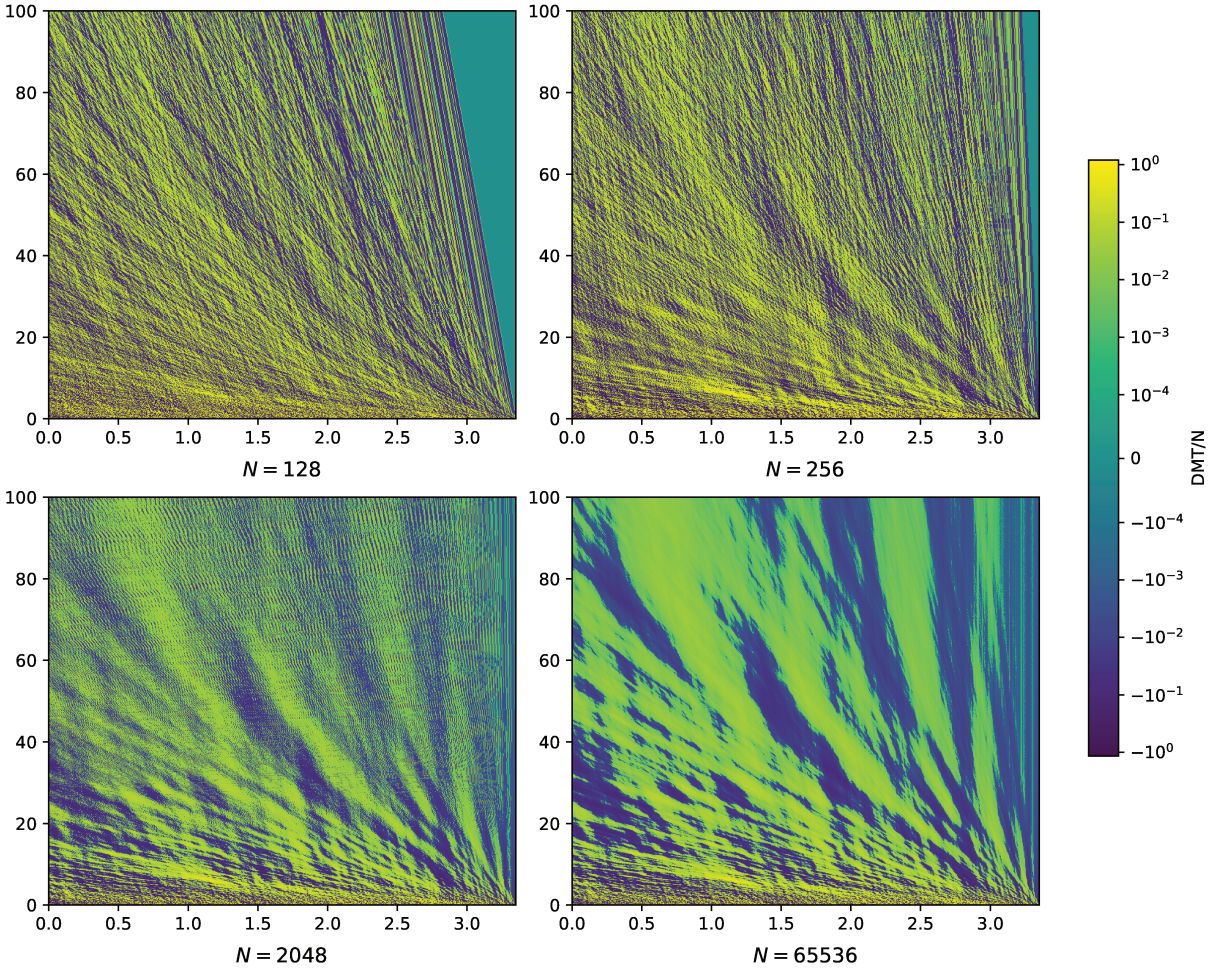


Figure 8: Impact of varying the parameter N, introduced in (7), on an example signal processed in a single DMT (no overlap, no streaming). $t_0@f_{max}$ is on the x axis, in seconds, and the DM is on the y axis, in parsec per cubic centimeters.
Notice how increasing N increases contrast and resolution, and reduces aliasing. Higher dispersion measures require bigger values of N.

2.3.4 Telemetry

This pipeline implements two kinds of telemetry: waveform storage trigger, and memory dumps for observability.

The first telemetry component is targeting an upcoming system in NenuFAR, which will feature very long ring buffers and will be dedicated to storing portions of the raw BHR signal at full resolution. Our pipeline is designed to trigger this system when a dispersed signal candidate is detected. After that, an in-depth inspection can be done offline to confirm or reject the candidate.

The second telemetry component concerns the pipeline's observability, which represents a challenge for any real-time system. It is especially needed during the first iterations of the development cycle, to help debugging and understanding what happens in the pipeline, but it can become a bottleneck when dealing with large amounts of data. In the case of our pipeline, we dump the contents of output GPU buffers after each main component. The naive approach of performing a memory copy from GPU to CPU and then writing to disk was putting pressure on the PCIe bus, interfering with the acquisition system and causing packet losses. To circumvent this issue, we used a DMA from GPU to disk, as enabled by the GPUDirect Storage API provided by Nvidia. This path is represented using dotted lines on Figure 3, and has been used to produce plots shown in section A.

2.3.5 Implementation details

The CuTe[45] library is used to describe tensor layouts for every algorithm implementation, in an effort to make this work more reusable.

This library enables the description of tensors with arbitrary shapes, that can be specified either at runtime or at compile time, through the use of `C++` templates. When possible, setting sizes at compile time may enable the compiler to optimize more aggressively.

Moreover, CuTe requires to describe the strides between elements of the tensor. This enables using seamlessly a row major, a column major, any other kind of arrangement. Our work relies heavily on the row major format, which enables "coalesced memory accesses" in our implementations, enabling optimized memory transfers on Nvidia hardware, However, thanks to the use of CuTe, this can be easily changed if new constraints arise. This is also useful to enable interoperability with work based on another memory arrangement.

3 Results

3.1 Data acquisition

On NenuFAR, the challenge was a full real-time, production ready pipeline, able to run during several hours. The current version matches these requirements, by acquiring NenuFAR's BHR streams without any packet loss in a continuous survey.

An analysis over multiple use cases — presented more thoroughly in a previous article[28] — shows the versatility of this data acquisition, and good performance both in terms of latency, jitter and maximum data rate. This method is reported to be able to receive up to 99 Gbit/s without packet loss (without considering further processing stages).

These results are a first step towards data acquisition on upcoming giant telescopes, where data rates will reach several tens of terabits per second. The only missing building block in our work to scale to this range of data rate is work distribution among multiple NICs. However, building blocks in DPDK enable to offload this task to the NIC using standard approaches such as Receive Side Scaling (RSS).

3.2 Pipeline performance

We went through a performance analysis to characterize our implementations, and enable easier tuning of the pipeline. The computation time of our algorithms depends on some tunable pipeline parameters, and this characterization enables us to choose the best set of parameters to run our pipeline, maximizing SNR while keeping real-time performance.

3.2.1 Methodology

The algorithms were benchmarked over a specific parameter grid, described in Table 3

Parameter	Set of values
N_f	$\{k \cdot 192 \mid k \in \{1, ..., 4\}\}$
N_t	$\left\{2^k \mid k \in \{10, ..., 21\}\right\}$
N_{DM}	$\left\{2^k \mid k \in \{8, ..., 12\}\right\}$
$N_{t_{out}}/N_{t_{in}}$	$1, ..., 8$
N_{cand}	$\left\{2^k \mid k \in \{8, ..., 12\}\right\}$
N	$\left\{2^k \mid k \in \{8, ..., 12\}\right\}$
f_{min}	$\left\{10 + k\frac{50-10}{16-1}\,\text{MHz} \mid k \in \{0, ..., 15\}\right\}$
f_{max}	$\left\{70 + k\frac{85-70}{16-1}\,\text{MHz} \mid k \in \{0, ..., 15\}\right\}$
DM_{min}	$0\,\text{pc}\,\text{cm}^{-3}$
DM_{max}	$\left\{2^k\,\text{pc}\,\text{cm}^{-3} \mid k \in \{6, ..., 12\}\right\}$

Table 3: Parameter space

Exploring the full parameter space for each algorithm would take significant time. Over 25 million configurations would have to be tested, resulting in about a year worth of computations, under the hypothesis of a mean computation time of 1 s.

To speed this process up, we performed a random sampling of the parameter grid, and searched for execution time models. We compute the coefficient of determination R^2 to assess the fitness of our models.

3.2.2 Execution time models

We benchmarked our implementations on a Nvidia A100 GPU, and collected the execution models listed in Table 4. The resulting models follow closely the experimental measures, according to the coefficients of determination. We can use these models to find models for the relative execution times of our algorithm implementations compared to real-time constraints.

Comparing the execution time to the duration of the portion of signal being processed is interesting to assert whether we comply with the real-time execution constraint, and if we have remaining computation time available on the GPU.

Stage	Execution time model	R^2	Relative execution time model
Time integration	$4.92 \times N_f \times N_t$ ps	0.999 90	$9.61 \times 10^{-7} \times N_f$
MAD Clean	$72.52 \times N_f \times N_t$ ps	0.999 96	$14.16 \times 10^{-6} \times \frac{N_f}{N_{fold}}$
Rm Baseline	$9.63 \times N_f \times N_t$ ps	0.968 27	$1.88 \times 10^{-6} \times \frac{N_f}{N_{fold}}$
Rm Mean	$14.66 \times N_f \times N_t$ ps	0.999 99	$2.86 \times 10^{-6} \times \frac{N_f}{N_{fold}}$
DMT	$0.96 \times N_{DM} \times N_{t_{out}} \times N$ ps	0.998 17	$1.88 \times 10^{-7} \times \frac{N_{DM} \times N_{t_{out}} \times N}{N_{t_{in}} \times N_{fold}}$
Detection	$360 \times N_{DM} \times N_t$ ps	0.997 98	$70.28 \times 10^{-6} \times \frac{N_{DM}}{N_{fold}}$
Tight loop	$4.92 \times N_f \times N_t$ ps		$9.61 \times 10^{-7} \times N_f$
Medium loop	$178.95 \times N_f \times N_t$ ps		$34.95 \times 10^{-6} \times \frac{N_f}{N_{fold}}$
Loose loop	$N_{DM}\,(0.96 \times N_{t_{out}} \times N + 360 \times N_{t_{in}})$		$\frac{N_{DM}}{N_{fold}}\left(70.28 + 1.88 \times 10^{-7} \times \frac{N_{t_{out}} \times N}{N_{t_{in}}}\right)$

Table 4: Execution time models. Please refer to Figure 4 for a description of the tight, medium and loose loops.

On the most common configuration used, with $N_f = 192$, $N_{DM} = 4096$ and $N_{fold} = 1024$, the computation time is negligible compared to real time (0.1 %), which leaves ample room for other work as shown on Figure 9. This also leaves great promises for a scale up, at first on one GPU and to then to GPU clusters for other projects of real-time FRB detection.

As explained in section 2.3.1, the time resolution has a great impact on the probability of detection and is subject to tuning. In this version, these parameters are fixed to a compromise value, but we can imagine exploring this parameter space by running multiple instances of the

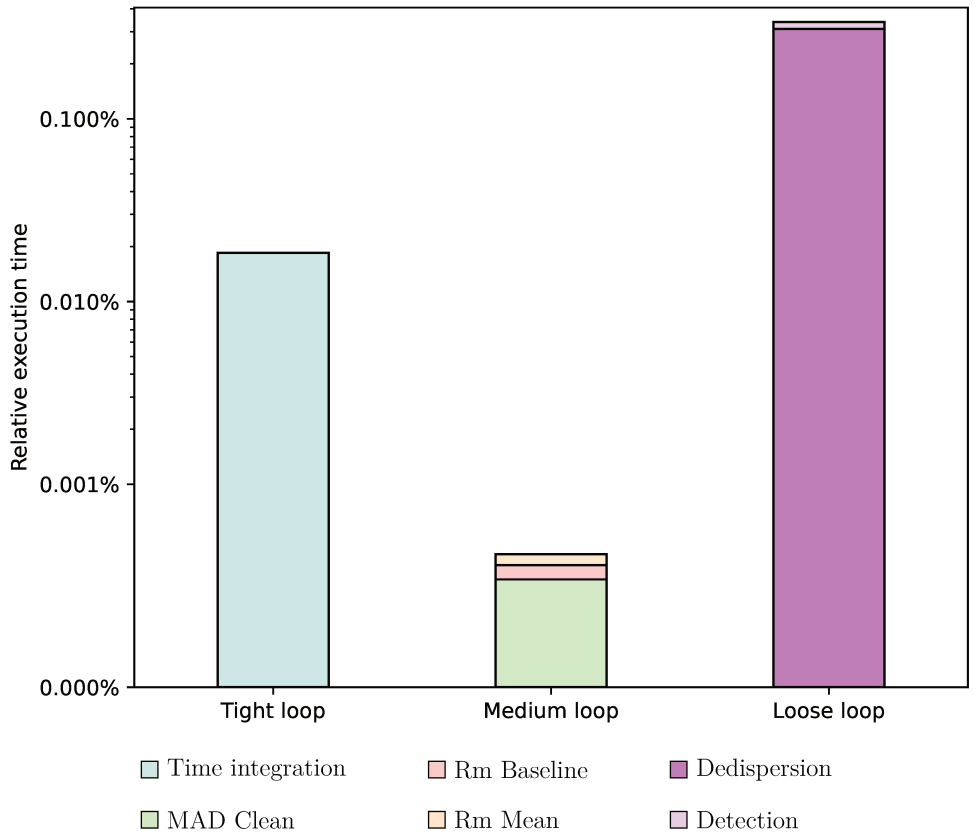


Figure 9: Ratio between computation time and critical time for each loop introduced in Figure 4, on the common configuration $N_f = 192$, $N_{DM} = 4096$, $N_{fold} = 1024$, $\frac{N_{t_{out}}}{N_{t_{in}}} = 4$ and $N = 1024$. The total GPU usage is of 0.36 % with this set of parameter and a Nvidia A100 GPU. On a more extreme case ($N_f = 768$, $N_{DM} = 4096$, $N_{fold} = 64$, $\frac{N_{t_{out}}}{N_{t_{in}}} = 4$ and $N = 8192$), the total GPU usage reaches 40 %. A tradeoff has to be chosen between computation time, instrument sensitivity and energy consumption.

pipeline with a different sample of parameters. This would increase the chance of detecting extreme radio transients, with little further effort.

3.2.3 Roofline model

We present a roofline model visualization (Figure 10) to categorize our implementations and help focusing next optimization efforts.

Using this roofline model, we notice that the implementations of all the pipeline components are memory bound; that is, the memory system is the bottleneck. This is expected, since GPUs are tuned for relatively complex operations (focus on matrix multiplication, $\mathcal{O}(n^3)$) while most of our algorithms, especially during preprocessing, are of linear complexity. Knowing this is encouraging towards the use of lower-end GPUs, most of the time featuring less compute resources but similar memory systems for a given GPU generation. This would enable lower production costs for a similar result, in the context of FRB detection.

The exception is the Dedispersion Measure Transform, as the implementation we developed has cubic complexity,[39] and should be compute bound while approaching the GPU's maximum floating point performance (the horizontal blue line). Our hypothesis is that the variation we introduced to support strongly dispersed signals, especially through equation (6), rely on a lot of integer arithmetic for offset computation. This makes the floating point roofline model representation a bad fit to characterize this implementation, as integer instructions are not taken into account. The proportion of integer instructions compared to floating point instructions is 6:1 for our implementation of dedispersion, whereas it is in the order of 1:10 for other components of the pipeline.

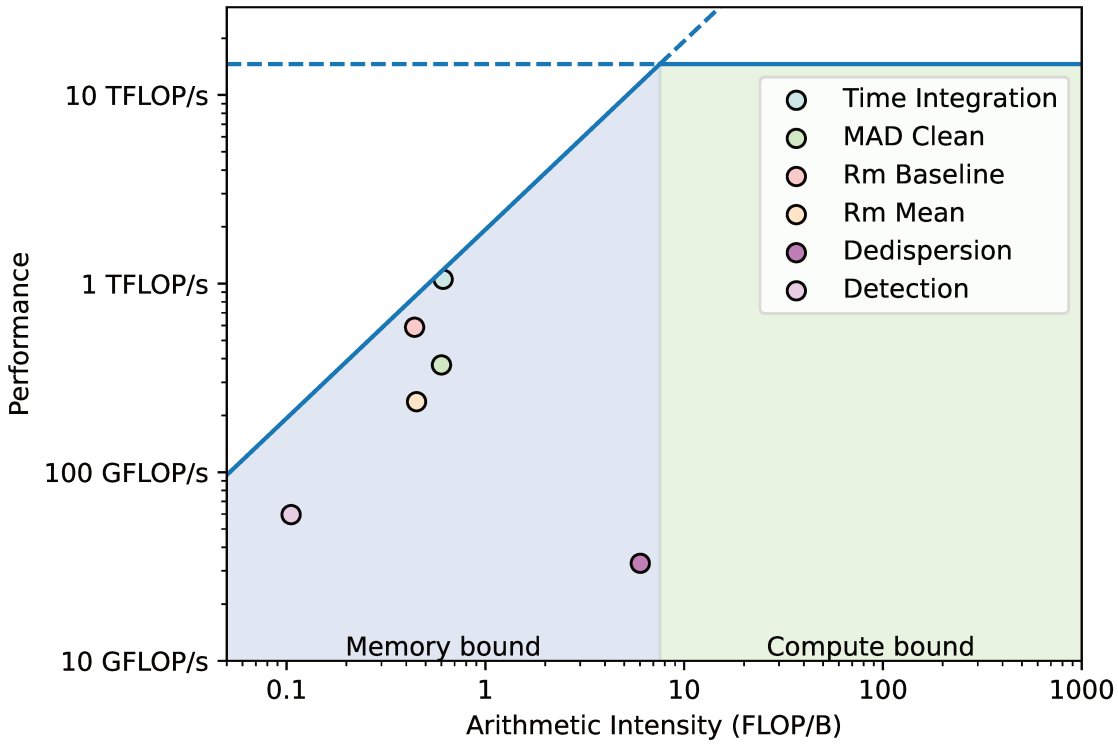


Figure 10: FP32 Roofline model. Metrics are measured on a problem size big enough to saturate the GPU, which is the case with the common configuration described in Figure 9. The results are the same with any configuration saturating the GPU, except for the Dedispersion, which depends on the ratio $\frac{N_{t_{out}}}{N_{t_{in}}}$. Here, $\frac{N_{t_{out}}}{N_{t_{in}}}$ is set to 4, which is the case in the configuration used in production.

3.3 Validation on known pulsars

We validated this pipeline on pulsar data, which shares similar characteristics than FRBs: short, intense pulses featuring dispersion. Table 5 lists the pulsars successfully detected on NenuFAR using this pipeline. Uncertainties are given with a confidence level of 95 %, and experimental results are compared to catalog data from PSRCAT.[46,47] Our pipeline, using a single-pulse search mode, gets results within a 0.26 % relative error margin from the catalogue for DM values, and 0.71 % for period values.

Source	Parameter	Measured	Reference		SNR
B1919+21	DM ($\mathrm{pc\,cm^{-3}}$)	12.47(26)	12.443 99(63)	0.26 %	4.8(13)
	Pulses	1649	2692	61.3 %	
B0809+74	DM ($\mathrm{pc\,cm^{-3}}$)	5.73(32)	5.750 66(48)	0.23 %	4.4(13)
	Pulses	1306	2786	46.9 %	
B0834+06	DM ($\mathrm{pc\,cm^{-3}}$)	12.84(26)	12.8640(4)	0.17 %	4.4(13)
	Pulses	530	2785	18.8 %	

Table 5: Detected pulsars. The column containing percentages describes relative errors for DM measurements, and detection percentages for pulses measurements.

4 Future work

Many new features, algorithms and technologies are planned to be integrated in this pipeline, that will serve as a sandbox for many topics.

4.1 Real-time data acquisition

Efforts to remove the CPU from the critical path can be brought even further. In the current version, the CPU is still the data acquisition orchestrator: it triggers the DMA from NIC to GPU. Even though this mode of operation can receive 10 Gbit/s without any packet loss, and has been showed[26] to scale up to 99 Gbit/s, this mode requires CPU-GPU communication and synchronization, which increases the complexity of the system and will eventually become a bottleneck while scaling up to larger bandwidths, because of the additional PCIe traffic.

A new technology introduced by Nvidia through its closed-source DOCA SDK,[48] GPUNetIO,[49,50] makes it possible to trigger the DMA from the GPU directly. In this way, CPU can be completely also removed from the control path and the GPU is fully independent to do both packet processing and network control. This is one step further to bypass the CPU, to reduce latencies, and to simplify this data acquisition method.

Another alternative would be to offload packet processing to the NIC, since the Nvidia BlueField-2 is a Data Processing Unit (DPU), an emerging category of Smart NICs able to run code on embedded CPU cores. In the case of our BlueField-2, 8 ARM cores are available, that could possibly run a packet processing task. The DMA could then occur between these ARM cores and the GPU, without any action from the host CPU. The advantage of this scheme is that no packet processing kernel has to run on the GPU anymore, which simplifies even more the data acquisition system. The open questions about this approach relate to the ability to perform DPU-GPU DMA, and synchronization issues. The DOCA SDK encompasses the DPU use case, and may provide support to these issues.

However, both of these options are strongly tied to the DOCA SDK, which raises important concerns about the portability of these approaches. One of the goals of the DPDK is to be portable accross different vendors, even though support for functionalities can vary greatly from one vendor to another. Conversely, the DOCA SDK is limited to Nvidia hardware.

4.2 Real-time processing

This first iteration contains basic implementations of the many stages of a classical FRB search pipeline, adapted to the low observation frequencies of NenuFAR. New algorithms are to be implemented, in order to further increase the SNR. This will increase the sensitivity of this detection pipeline, and help finding the first FRBs on NenuFAR.

4.2.1 Time integration

Currently, only a rectangular window is implemented for the Short-time Fourier Transform. This window is especially subject to spectral leakage,[51] which can be reduced by using other windows such as the Hamming or Hann windows. These windows can help increasing the SNR.

The time averaging part of time integration could also be moved to detection, to support detecting pulses of various durations more reliably.

4.3 Preprocessing

More advanced methods have been implemented for RFI mitigation.[52,53] Implementing these methods will help to reduce even further the impact of RFIs in our signal, to increase the level of confidence of our detection and increase SNR at the same time.

4.4 Dispersion Measure Transform

Only the naive Bruteforce DMT is currently implemented as a backend for Streaming DMT. The state-of-the-art Fast DMT[40] will be implemented to even further reduce the complexity of this step. This has a lower priority, since the computation time is not an issue at this stage, but this will be needed to reduce the factor of pure time integration in exchange of STFT. The latter increases the number of frequency channels, which will increase the computation time.

Implementing coherent, or semi-coherent dedispersion will also be interesting at some point, although it will be more difficult to implement them using the Streaming DMT approach, given these other dedispersion methods do not rely on simple sums, but shifts in the Fourier domain. However, this may be a crucial step to FRB detection on NenuFAR, as incoherent dedispersion loses becomes insensitive on shorter pulses.[40] The incoherent dedispersion is insensitve for $t_p \ll t_{inc}$, where t_p is the duration of a pulse and t_{inc} is the minimal dispersion smearing given by:[40]

$$t_{inc} = 3.13 \times 10^{-5} \left(\frac{DM}{100\,\mathrm{pc\,cm^{-3}}} \right)^{1/2} \left(\frac{f_{min}}{\mathrm{GHz}} \right)^{-1} \left(\frac{f_{max}}{\mathrm{GHz}} \right)^{-1} \left(\frac{f_{min} + f_{max}}{2\,\mathrm{GHz}} \right)^{1/2} \mathrm{s}$$

In the case of NenuFAR, a reasonable upper bound is $t_{inc} = 7.3\,\mathrm{ms}$, for $f_{min} = 40\,\mathrm{MHz}$, $f_{max} = 85\,\mathrm{MHz}$ and $DM = 1000\,\mathrm{pc\,cm^{-3}}$. In the context of NenuFAR, due to interstellar and intergalactic scattering, as well as intra-channel dispersion, FRB durations are of the order of 100 ms. Hence, incoherent dedispersion will not be a major concern for the detection FRBs.

4.5 Detection

More robust statistics such as the Median Absolute Deviation (MAD) or the Interquartile Range (IQR) will be implemented to replace standard deviation, which is more impacted by outliers. This will help reducing the number of false positives, and as such the specificity.

Clustering methods taking into account the butterfly shape of the single pulses in the output of the DMT might also be interesting to deploy, to even further reduce the number of false negatives.

4.6 SNR study

The execution time of the components of our proposed pipeline has been studied, but this has not been linked to SNR improvement. Injection of mock FRBs have been proposed[54] to facilitate SNR estimation, by expecting a specific FRB at a given t_0 and DM. Doing so will at the same time help checking that further improvements do not degrade detection performance.

4.7 Portability

Most of this work relies on Nvidia NICs and GPUs. The codebase is written using Nvidia's CUDA language, and DMA features are eased by Nvidia components. The promise of DPDK is to be portable across multiple vendors, but available functionalities vary greatly depending on the actual hardware being used; as such, porting to another NIC might not be trivial. Moreover, CUDA code would have to be ported to another language to use this work on other accelerators. AMD's HIP language provide a HIPIFY script to transpile from CUDA to HIP, and could help porting this pipeline to AMD hardware very easily. The SYCL language looks like another interesting target to replace CUDA and increasing portability, as SYCL code can be run on heterogeneous targets, including Nvidia and AMD GPUs, but also CPUs and FPGAs. Finally, the SPDK could be used to replace GPUDirect Storage.

4.8 COSMIC Framework integration

An additional option to increase portability, reusability and flexibility of this code will be to integrate it in the COSMIC[55] framework. This framework facilitates the development of reconfigurable pipelines, by partitioning each block of a pipeline in a Business Unit (BU). These BUs communicate with each other through shared memory, and can be replaced at runtime. As such, adding new functionalities or algorithms to the existing pipeline can be made easier using this framework.

Moreover, this framework ships with a Python wrapper, that could be used to easily control the pipeline at runtime and to improve its observability.

5 Conclusion

The first iteration of a Fast Radio Burst detection pipeline for NenuFAR has been completed. This version is able to acquire the BHR streams sent by the LaNewBa beamformer, perform time integration, apply a simple preprocessing (simple RFI mitigation, signal normalization), compute the incoherent dedispersion using a new Streaming DMT approach, and detecting single pulses, in real-time. A dedicated storage unit can then be triggered, to exclusively store parts of the signal containing potential radio transients.

This pipeline relies on a new approach to data acquisition, based on Direct Memory Access from NIC to GPU, removing almost completly the CPU from the critical path. It also relies on a new approach for incoherent dedispersion, proposed as the Streaming Dispersion Measure Transform, which removes the need for overlap and help reducing memory footprint, which is crucial for FRB detection at low frequency, where dispersive delays can typically reach several minutes.

These two new approaches, as well as the rest of the GPU implementations, have been validated on single pulse blind search over multiple known pulsars, with great success.

Considering the low GPU usage of this first version, we can imagine adding more complex algorithm to iteratively increase the sensitivity of this pipeline.

Disclosures

No conflict of interest are to be declared.

Acknowledgments

We sincerely thank the French Region Île-de-France for funding this work through the Paris Region PhD program (DIM ACAV+ #20007183).

We are also grateful to the Observatoire Radioastronomique de Nançay, especially Cédric Viou and Emmanuel Thétas for their help deploying this work on site and Dr Louis Bondonneau for his counselling about FRB detection, as well as Dr Jean-Mathias Grießmeier (LPC2E, USN) and Dr Valentin Decoene (Nantes Université) for approving our piggybacking on the LT03 and LT05 observation programs.

Many thanks as well to Stephen Jones (Nvidia), for his ever accurate insights about the intricacies of CUDA distilled during our meetings.

Finally, thanks should also go to Elena Agostini (Nvidia) for her implication in DPDK and her work to enable NIC to GPU DMA, as well as the collaboration we had about this matter.

Code, Data, and Materials Availability

We provide the code of the pipeline presented in this paper under LGPL: https://gitlab.obspm.fr/jplante/NenuFRB-rt.

The output of this pipeline is not yet publicly available, but may be published in future articles.

References

1 E. Petroff, J. W. T. Hessels, and D. R. Lorimer, "Fast radio bursts at the dawn of the 2020s," *The Astronomy and Astrophysics Review* **30**, 2 (2022).

2 E. Petroff and O. Yaron, "Fast Radio Burst Catalogue on the TNS," *Transient Name Server AstroNote* **160**, 1 (2020).

3 Z. Pleunis, D. Michilli, C. G. Bassa, *et al.*, "Lofar detection of 110–188 mhz emission and frequency-dependent activity from frb 20180916b," *The Astrophysical Journal Letters* **911**, L3 (2021).

4 Bondonneau, L., Grießmeier, J.-M., Theureau, G., *et al.*, "Pulsars with nenufar: Back-end and pipelines," *A&A* **652**, A34 (2021).

5 P. Zarka, L. Denis, M. Tagger, *et al.*, "The low-frequency radio telescope nenufar," in *GASS*, URSI (2020).

6 A. Loh and the NenuFAR team, "nenupy: a python package for the low-frequency radio telescope nenufar," (2020).

7 P. Zarka, "Nenufar beamforming time-frequency data," in *2nd NenuFAR User Workshop*, (2021).

8 L. Bondonneau, "Nenuraw." https://github.com/louisbondonneau/NenuRaw.

9 B. H. Leitao, "Tuning 10gb network cards on linux," in *Proceedings of the 2009 Linux Symposium*, 169–185, Citeseer (2009).

10 P. Emmerich, M. Pudelko, S. Bauer, *et al.*, "User space network drivers," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 1–12 (2019).

11 J. S. Leger, "Myth-busting dpdk in 2020," *AvidThink* (2020).

12 L. Rizzo, "netmap: A novel framework for fast packet I/O," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 101–112, USENIX Association, (Boston, MA) (2012).

13 D. Perret, M. Lainé, J. Bernard, *et al.*, "Bridging fpga and gpu technologies for ao real-time control," in *Adaptive Optics Systems V*, **9909**, 1364–1374, SPIE (2016).

14 M. Amiri, K. Bandura, P. Berger, *et al.*, "The chime fast radio burst project: System overview," *The Astrophysical Journal* **863**(1), 48 (2018).

15 G. M. Callanan, "A gpu based x-engine for the meerkat radio telescope," Master's thesis, University of Cape Town (2020).

16 W. Dai and F. Wang, "Study on processing performance of a dpdk and gpu combined pulsar data reduction system," *International Journal of Mechatronics and Applied Mechanics* (6), 213–224 (2019).

17 E. Petroff, J. W. T. Hessels, and D. R. Lorimer, "Fast radio bursts," *The Astronomy and Astrophysics Review* **27**, 4 (2019).

18 S. M. Ransom, *New search techniques for binary pulsars*. PhD thesis, Harvard University, Massachusetts (2001).

19 K. W. Bannister, R. M. Shannon, J.-P. Macquart, *et al.*, "The detection of an extremely bright fast radio burst in a phased array feed survey," *The Astrophysical Journal Letters* **841**, L12 (2017).

20 M. P. Surnis, D. Agarwal, D. R. Lorimer, *et al.*, "Greenburst: A commensal fast radio burst search back-end for the green bank telescope," *Publications of the Astronomical Society of Australia* **36**, e032 (2019).

21 M. C. Bezuidenhout, E. Barr, M. Caleb, *et al.*, "MeerTRAP: 12 Galactic fast transients detected in a real-time, commensal MeerKAT survey," *Monthly Notices of the Royal Astronomical Society* **512**, 1483–1498 (2022).

22 D. J. Champion, E. Petroff, M. Kramer, *et al.*, "Five new fast radio bursts from the HTRU high-latitude survey at Parkes: first evidence for two-component bursts," *Monthly Notices of the Royal Astronomical Society: Letters* **460**, L30–L34 (2016).

23 W. Farah, C. Flynn, M. Bailes, *et al.*, "Five new real-time detections of fast radio bursts with UTMOST," *Monthly Notices of the Royal Astronomical Society* **488**, 2989–3002 (2019).

24 I. P. Kravtsov, V. V. Zakharenko, I. Y. Vasylieva, *et al.*, "Decameter pulsars and transients survey of the northern sky. observations and data processing," in *2016 9th International Kharkiv Symposium on Physics and Engineering of Microwaves, Millimeter and Submillimeter Waves (MSMW)*, 1–4 (2016).

25 K. Mikhailov and A. Sclocco, "The apertif monitor for bursts encountered in real-time (amber) auto-tuning optimization with genetic algorithms," *Astronomy and Computing* **25**, 139–148 (2018).

26 J. Plante, D. Gratadour, L. Bondonneau, *et al.*, "A novel frb detection pipeline for nenufar," (2021).

27 E. Agostini, "Boosting inline packet processing using dpdk and gpudev with gpus," (2022). https://developer.nvidia.com/blog/optimizing-inline-packet-processing-using-dpdk-and-gpudev-with-gpus/.

28 J. Plante, D. Gratadour, L. Matias, *et al.*, "A high performance data acquisition on cots hardware for astronomical instrumentation," in *Software and Cyberinfrastructure for Astronomy VII*, **12189**, 323–332, SPIE (2022).

29 S. Ransom, "PRESTO: PulsaR Exploration and Search TOolkit." Astrophysics Source Code Library, record ascl:1107.017 (2011).

30 C. N. P. G. Arachchige and L. A. Prendergast, "Confidence intervals for median absolute deviations," *arXiv e-prints* , arXiv:1910.00229 (2019).

31 A. R. Offringa, A. G. de Bruyn, M. Biehl, *et al.*, "Post-correlation radio frequency interference classification methods," *Monthly Notices of the Royal Astronomical Society* **405**(1), 155–167 (2010).

32 L. Li, P. Gaiser, M. Bettenhausen, *et al.*, "Windsat radio-frequency interference signature and its identification over land and ocean," *IEEE Transactions on Geoscience and Remote Sensing* **44**(3), 530–539 (2006).

33 G. M. Nita, A. Keimpema, and Z. Paragi, "Statistical discrimination of rfi and astronomical transients in 2-bit digitized time domain signals," *Journal of Astronomical Instrumentation* **08**(01), 1940008 (2019).

34 J. Taylor, N. Denman, K. Bandura, *et al.*, "Spectral Kurtosis-Based RFI Mitigation for CHIME," *Journal of Astronomical Instrumentation* **8**, 1940004 (2019).

35 L. Hui, D. Yu-jun, L. Xiang-ru, *et al.*, "The sumthreshold method for radio frequency interference detection," *Chinese Astronomy and Astrophysics* **46**(3), 277–296 (2022).

36 J. Akeret, S. Seehars, C. Chang, *et al.*, "Hide & seek: End-to-end packages to simulate and process radio survey data," *Astronomy and Computing* **18**, 8–17 (2017).

37 J. Akeret, C. Chang, A. Lucchi, *et al.*, "Radio frequency interference mitigation using deep convolutional neural networks," *Astronomy and Computing* **18**, 35–39 (2017).

38 D. Czech, A. Mishra, and M. Inggs, "A cnn and lstm-based approach to classifying transient radio frequency interference," *Astronomy and Computing* **25**, 52–57 (2018).

39 B. R. Barsdell, M. Bailes, D. G. Barnes, *et al.*, "Accelerating incoherent dedispersion," *MNRAS* **422**, 379–392 (2012).

40 B. Zackay and E. O. Ofek, "An accurate and efficient algorithm for detection of radio bursts with an unknown dispersion measure, for single-dish telescopes and interferometers," *The Astrophysical Journal* **835**(1), 11 (2017).

41 M. Ester, H.-P. Kriegel, J. Sander, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, *KDD'96*, 226–231, AAAI Press (1996).

42 D. Pang, K. Goseva-Popstojanova, T. Devine, *et al.*, "A novel single-pulse search approach to detection of dispersed radio pulses using clustering and supervised machine learning," *Monthly Notices of the Royal Astronomical Society* **480**, 3302–3323 (2018).

43 D. Agarwal, K. Aggarwal, S. Burke-Spolaor, *et al.*, "FETCH: A deep-learning based classifier for fast transient classification," *Monthly Notices of the Royal Astronomical Society* **497**, 1661–1674 (2020).

44 L. Connor and J. van Leeuwen, "Applying deep learning to fast radio burst classification," *The Astronomical Journal* **156**, 256 (2018).

45 NVIDIA, "Cutlass." https://github.com/NVIDIA/cutlass.
46 "Atnf pulsar catalogue." https://www.atnf.csiro.au/research/pulsar/psrcat/.
47 R. N. Manchester, G. B. Hobbs, A. Teoh, *et al.*, "The Australia Telescope National Facility Pulsar Catalogue," *AJ* **129**, 1993–2006 (2005).
48 "Doca sdk." https://developer.nvidia.com/networking/doca.
49 E. Agostini, "Nvidia doca gpunetio," (2022). https://developer.nvidia.com/blog/inline-gpu-packet-processing-with-nvidia-doca-gpunetio/.
50 NVIDIA, *NVIDIA DOCA GPUNetIO Programming Guide*.
51 P. Podder, T. Z. Khan, M. H. Khan, *et al.*, "Comparative performance analysis of hamming, hanning and blackman window," *International Journal of Computer Applications* **96**(18), 1–7 (2014).
52 Fridman, P. A. and Baan, W. A., "Rfi mitigation methods in radio astronomy," *A&A* **378**(1), 327–344 (2001).
53 G. M. Nita and G. Hellbourg, "A cross-correlation based spectral kurtosis rfi detector," in *2020 XXXIIIrd General Assembly and Scientific Symposium of the International Union of Radio Science*, 1–4 (2020).
54 V. Gupta, C. Flynn, W. Farah, *et al.*, "Estimating fast transient detection pipeline efficiencies at UTMOST via real-time injection of mock FRBs," *Monthly Notices of the Royal Astronomical Society* **501**, 2316–2326 (2020).
55 F. Ferreira, A. Sevin, J. Bernard, *et al.*, "Hard real-time core software of the AO RTC COSMIC platform: architecture and performance," in *Adaptive Optics Systems VII*, L. Schreiber, D. Schmidt, and E. Vernet, Eds., **11448**, 1144815, International Society for Optics and Photonics, SPIE (2020).

List of Figures

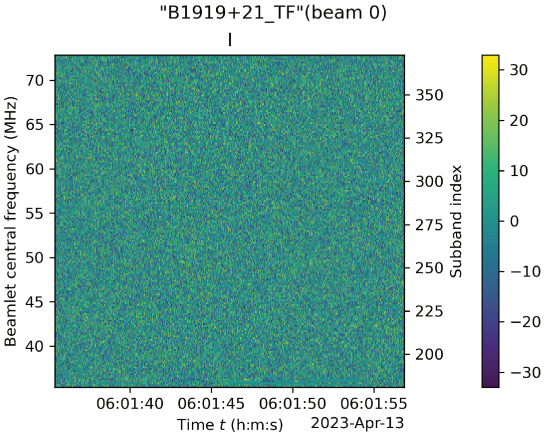
1 Network configuration overview
2 Thetis' PCIe topology summary
3 FRB detection pipeline overview
4 Execution timeline
5 RFI mitigation methods comparison
6 Preprocessing chain
7 Streaming DMT applied on simulated input
8 Impact of varying N
9 Ratio between computation time and critcal time for each loop
10 FP32 Roofline model
11 B1919+21
12 B0809+74
13 B0834+06

List of Tables

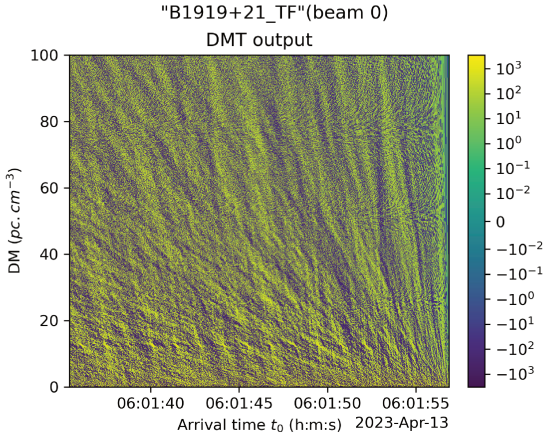
1 radio telescopes and their FRB detection pipelines
2 Server configuration: Thetis
3 Parameter space
4 Execution time models
5 Detected pulsars

Appendix A: Details on detected pulsars

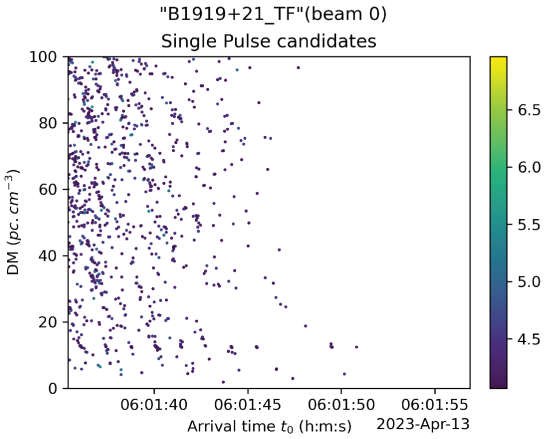
In this section are listed fractions of signal showing detection of the different pulsars listed in Table 5.



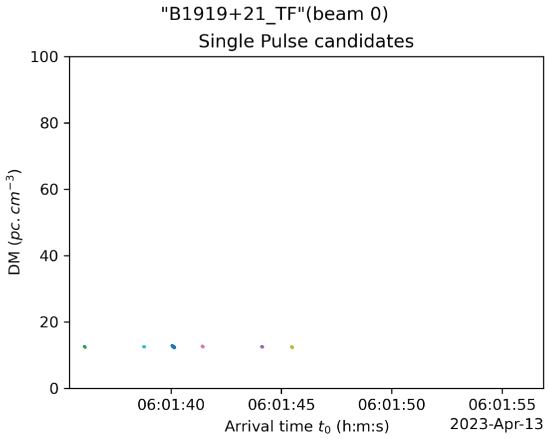
(a) Stokes I after preprocessing



(b) DMT space

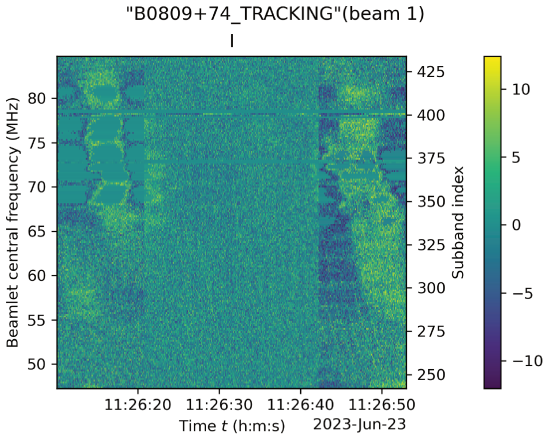


(c) FRB candidates

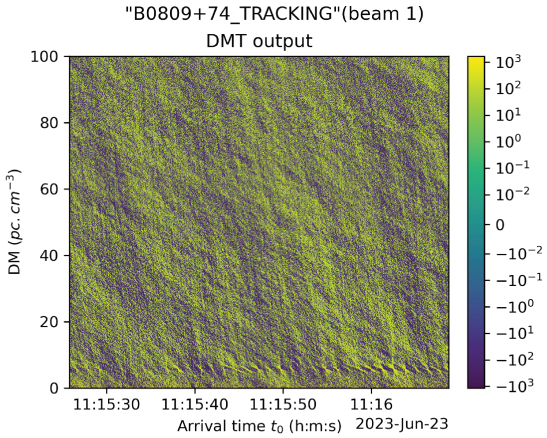


(d) Clustered candidates

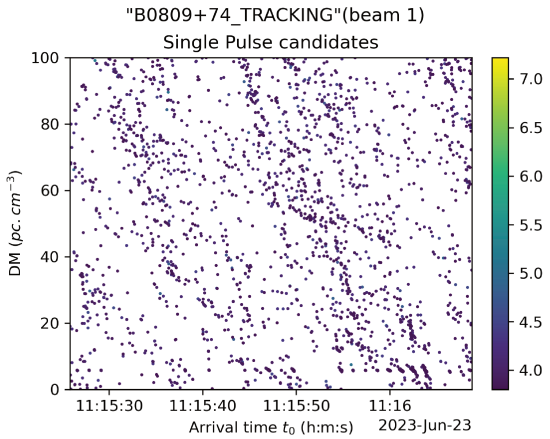
Figure 11: B1919+21



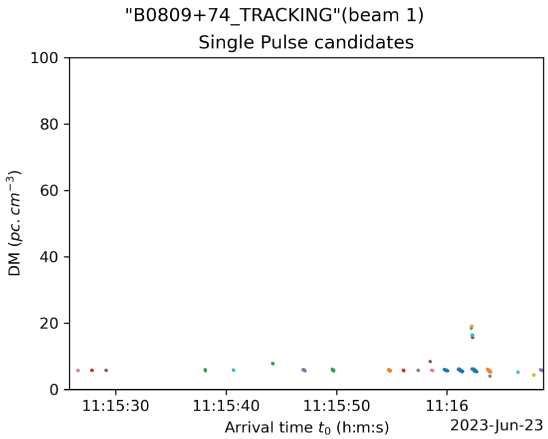
(a) Stokes I after preprocessing



(b) DMT space

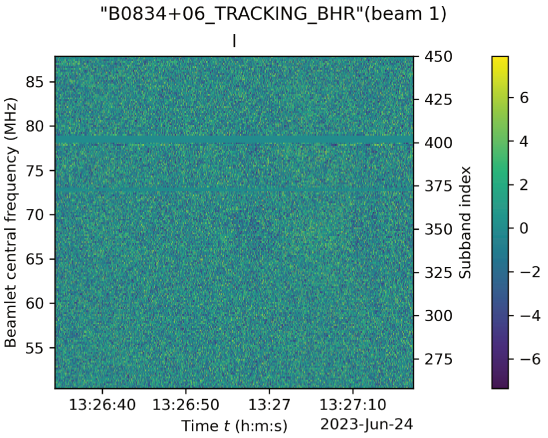


(c) FRB candidates

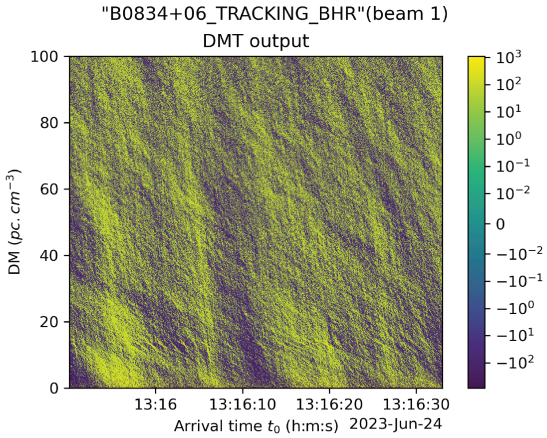


(d) Clustered candidates

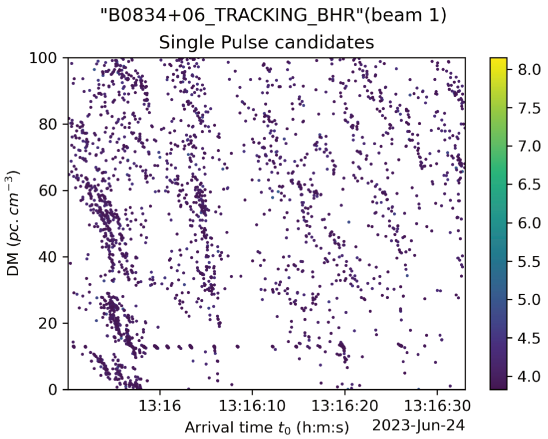
Figure 12: B0809+74



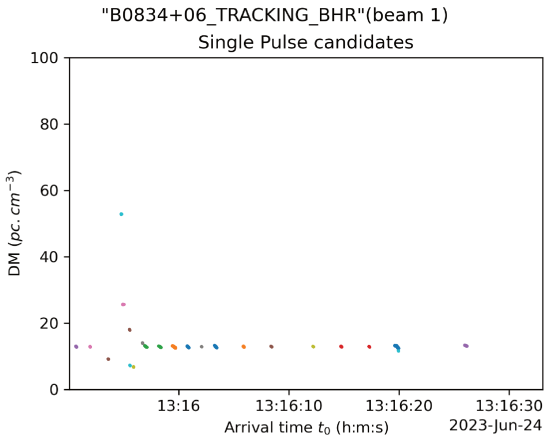
(a) Stokes I after preprocessing



(b) DMT space



(c) FRB candidates



(d) Clustered candidates

Figure 13: B0834+06

Chapter 7

Interesting connections between radar and radioastronomy

Even if these two domains can seem unrelated at first, they share some similarities, that can lead to interesting science, both in terms of algorithm design and software implementation. As an example, the simplest connection is that both observe similar frequencies. Interestingly, this has lead to a first, historical connection between both domains, as the first French radiotelescope, built in Nançay, was composed of two salvaged German radars, the Würzburg radars.

Because of the scope of this work, we did not fully exploit the connections listed in this section, but this can provide interesting ideas for future software reuse and algorithm development both for radiotelescopes and radar systems.

In this section, we provide a list of observed examples of connections between radar systems and radioastronomy. This list may not be exhaustive.

7.1 Beamforming

The first, and probably most blatant connection is related to the beamforming stage, which is a crucial step in interferometers. All Thales radars, as well as many radiotelescopes such as NenuFAR or SKA use the principle of interferometry in order to increase the aperture size (resolution) and collecting surface (sensitivity) cheaply, as well as to enable a custom emission/reception profile for the system. This is the exact same operation in both domain, albeit at different scales.

However, in both applications considered in this thesis, beamforming was done prior to acquisition on GPU, using arrays of FPGAs placed very near to the antennas. This makes it possible to achieve this compute intensive task in real-time, and to reduce the data rate transmitted from antennas to the backend computing system. As such, we did not have much interaction with this part of the systems.

Nevertheless, taking advantage of the high compute intensity of GPUs, we could imagine a GPU-powered beamforming, enabling many new features, such as the formation of rather wide beams for rough detection in a first iteration, and then much thinner beams in case of detection to increase resolution. This kind of on-demand beamforming could be very valuable and could be much more energy efficient if leveraging emerging technologies such as tensor cores. Work in this direction has already been realized in the context of radioastronomy[8, 55].

On NenuFAR, even though there is no active project towards a similar GPU-powered beamforming, the same advantages would apply. Additionally, each observation program could form beams their own way, instead of requesting a specific set of beams to LaNewBa, and as such virtually remove the limit on observation time, as long as objects studied are visible in the sky.

However, in the context of NenuFAR, this would come at the cost of a greatly increased energy consumption as the beamforming stage would be applied independently and repeatedly for each observation program. This would also require a significant upgrade in NenuFAR's infrastructure, as the existing 10 Gbit/s link would not be enough to transmit the signal from all mini stations.

Keeping compatibility with LOFAR should also be a priority, and a major concern in such an upgrade.

7.2 Feature extraction

In both primary radar SP chain and FRB detection pipeline, we try to detect a non-repeating, dispersed signal. Indeed, the range cell migrations of our MMHRSP is similar to the dispersion effect we observe on FRBs, even though they follow slightly different equations (polynomial vs inverse quadratic). From the point of view of the developer though, the constraints are very similar.

Moreover, a parallel can be seen between the feature extraction techniques used in the context of this thesis and other algorithms:

- Hough Transform: Used in image processing, generally for line or ellipse detection, but defined for any curve, including polynomials or inverse quadratics.
- Radon Transform: Used in computed tomography scans, especially in its inverse form (the direct transform being generally performed by the action of rotating X-rays over a patient's body). The direct Radon Transform is the continuous version of the Hough Transform.
- Keystone Transform: Designed specifically to correct range migrations before DF, for radar systems, without a known target velocity value.

We understand that the Dispersion Measure Transform (DMT) is a special case of Hough Transform, detecting inverse quadratics parametrized by two values, t_0 and DM. Optimizations might be applied one way or another. Especially, no quasilinear complexity algorithm was found in the state of the art of Hough Transform, although the Fast Dispersion Measure Transform (FDMT) algorithm enables such $\mathcal{O}(n \log n)$ complexity. This might not be possible because of the variety of curves covered by the Hough Transform, and goes out of scope of this document, but may be worth looking into.

Moreover, the Keystone Transform is seemingly an analog to coherent dedispersion, albeit without the need for multiple hypotheses. Again, this goes out of scope, but could provide a much lighter solution to apply coherent dedispersion, which is impractical because of the very long and expansive FFTs that must be applied in the current algorithm.

Finally, in our approach to FRB detection (following state-of-the-art), we relied on incoherent dedispersion, and as such did not use phase information, dropping a part of the valuable information received from the sky. Approaches similar to Doppler Filtering, based on matched filters, may be interesting to apply to FRB detection, in order to better discriminate them from noise. However, it seems difficult, based only on the current knowledge of FRBs, to design matched filters. Dispersion and scattering are a first difficulty, although these mechanisms are well understood. But some FRBs are not received on the whole band, and the emission mechanism is not fully understood either, leaving two unknowns regarding feasibility.

7.3 Adaptive thresholding

Finally, in both applications, a final stage of thresholding is used in order to extract the position of peaks, representing potential targets, from the whole signal.

As developed in previous sections, this step is crucial in the precision of the application. A value of threshold too low will lead to many false positives, and conversely a threshold too high will lead to many false negatives. Moreover, the background noise between the peaks we want to detect does not necessarily have the same statistics, depending on the time of day, jamming, *etc.*

The problem we have to solve is very similar in both cases, except for the exact noise statistics. Nevertheless, it is interesting to notice that the different CFARs are more advanced, and could be used in place of the current approach relying on median and Median Absolute Deviation (MAD) for FRB detection, with the goal to increase precision.

Due to privacy policies of Thales, we were not able to reuse the implementations of CA-CFAR and OS-CFAR for FRB detection and did not take the time to reimplement it, but this is left as future work as it could provide improvements in detection quality.

Chapter 8

Future work

8.1 Alternatives to CUDA

The CUDA language is a wonderful tool for high-performance computing, and more precisely GPU computing. It benefits from a good documentation and support, many high-quality libraries that make it possible to unleash the full power of increasingly complex GPU, and a great compiler. This language was historically among the first to make GPU programming feasible at large scale.

However, it is a proprietary framework controlled by Nvidia, and supporting only Nvidia targets. Until now, these targets were only GPUs, but with the acquisition of Mellanox in 2022 and the popularization of Smart NICs and DPUs, as well as the release of DOCA, Nvidia is taking over the high-performance networking domain too. By building our systems around Nvidia solutions, we adopt a short-term vision: very good performance, results and development time now, but a risk of transition in the future that could mean a complete rewrite.

Such a transition could be motivated by multiple factors, including the following (among others), in descending order of likeliness:

- Excessive rise of costs of Nvidia hardware compared to that of other vendors: after building a dependency, a increasing costs could be beneficial for the company, taking advantage of a large community of vendor-locked consumers
- Desire of using French or European GPUs to increase sovereignty. However, as of 2023, no member of the European Union is developing a GPU publicly. However, the European Processor Initiative (EPI) is an example of initiative going in this direction, albeit only with CPUs and a new kind of accelerator called EPAC targeted at AI inference.
- Choice of another type of accelerator. This could be a long-time contender in the domain of HPC such as FPGA or Application Specific Integrated Circuit (ASIC), or innovative accelerators such as Kalray's Massively Parallel Processor Array (MPPA).
- Reduced competitiveness of Nvidia. Although it is the first GPU vendor today, AMD is catching up with innovative architectures and attractive prices[47]. Other vendors such as Intel have also begun developing their own GPUs.

In this section, we will present a non-exhaustive list of alternatives to CUDA that could have been used during this PhD, how well they compare with CUDA and whether they can do better or not regarding the shortcomings listed above.

8.1.1 OpenCL

OpenCL is historically the main competitor of CUDA, being the other language that helped program and popularize General Purpose GPU (GPGPU) since the middle of the 2000s.

OpenCL is a language developed first by Apple, and then by the Khronos group, a consortium of 170 companies developing standards for 3D computing. It is based on `C`, and only the standard

of the language is designed by Khronos, similarly to `C` or `C++`. Each vendor can then implement an OpenCL compiler for its hardware.

This is a double-edged approach: on the one hand, this theoretically makes it possible for OpenCL to support any hardware, and in practice, OpenCL can be successfully compiled to both CPU, GPU or FPGA targets. However, this is only possible if a compiler exists for the targeted hardware, and this is not always the case. Nvidia is one example, being very slow in its adoption of the standard.

There is no clear OpenCL ecosystem of libraries as of 2023. Many libraries seem to support OpenCL through a `C++` API, but does not provide access to an actual way to chain multiple GPU operations without interaction of the CPU to form a single pipeline, as we are doing in the context of this PhD.

8.1.2 ROCm / HIP

ROCm is the equivalent of the CUDA ecosystem, by AMD. It is a serious contender, with a complete choice of libraries. However, even if their base language, HIP, can be compiled to target Nvidia GPUs, the other libraries are AMD-only, falling in the same kind of drawbacks than the CUDA toolkit.

8.1.3 SYCL

SYCL is another language by Khronos, and based on modern `C++` (`C++17`). It is a superset of OpenCL, that can target all targets supporting OpenCL, but can also be transpiled to CUDA or ROCm in order to get the best optimizations possible.

Intel developed a massive ecosystem through its oneAPI toolkit, including FFT, matrix multiplication, and convolution implementations. However, it is not clear if non-Intel hardware is supported by these.

8.1.4 Others

8.1.4.1 Compute Shaders

Historically, and still for many people, the first goal of a GPU is to render a 3D scene to a display. This is usually done using a graphics library, such as OpenGL, Vulkan, Metal, Direct3D or WebGPU. The approach of these libraries is very different from what we have presented in this thesis. A graphics pipeline is composed of a number of *shaders*, such as the vertex shader or the geometry shader. These shaders are small programs, that instruct the GPU on how to render specific objects.

Recently, these graphics libraries began to include a new type of shader: the compute shader, which provide similar features than CUDA. This approach gives the maximum portability, but performance and feature set may not be ideal.

8.1.4.2 OpenMP / OpenACC / OpenHMMP

These three `C++` extensions are sets of `#pragma` directives that enable very easy parallelization, mainly through parallel for and parallel reduce operations. These technologies emerged with multicore CPUs, and are slowly adding GPU target capabilities. These approaches generally enable a very fast development cycle, at the cost of performance. They can also lead to very complicated code for algorithms requiring more than a couple of directives, and as such more difficult code to maintain in the end.

8.1.5 Going to a higher level

When developing a high-performance application, it is tempting to write very low-level code. Given the current tooling, this is a good tradeoff between resulting performance and development time, and even brings a non negligible level of satisfaction, due to the complexity of what has been achieved from the ground up. This is what has been done during this PhD, at various levels, and

is very justified in some cases, because of the lack of maturity and vision of some libraries, but this is strongly detrimental.

Consider the implementation presented in paragraph 5.2.5.2.4: it features very high performance and is very interesting, but is extremely low-level, and as such can only be used on one specific GPU model, and one data type.

However, using higher level abstractions, as explained in Section 5.2.10.2, can provide much more versatility. In this specific case, this was not done because CuTe is a very recent and complex library, with a very hard to follow documentation. Other reasons for keeping low-level code are preconceptions and/or pride.

This has not been explored during this PhD, but some work exists in this direction[40]. This kind of solution may be the best long-term solution, even if it does not deliver maximal performance today. Generally speaking, considering the increasingly diverse and complex accelerators ecosystem, it is more important than ever to focus on small, reusable and open-source building blocks implemented and optimized for a maximum number of targets, with a standard interface. The CuTe library is the one candidate for a unified interface, although it still depends on CUDA libraries as of writing. A huge investment would be required at first to provide the initial tooling, but it could result in overall great performance and efficient hardware usage, while keeping development time and headaches as low as possible. Other alternatives are presented below.

8.1.5.1 Python

Similarly as a few decades ago when some programmers refused to code in `C` or `C++` because it would always be slower than their manually and carefully crafted assembly code, developers feared interpreted languages such as Python compared to compiled language, because of performance issues. This is right when writing performance code directly in Python, but libraries such as Dask, Numba and Nvidia RAPIDS are quickly making this stereotype wrong. By being collections of very high-performance code optimized by teams of hundreds of specialists and updated with every new hardware, this has the potential to deliver much more performance that carefully hand-written CUDA code, as long as the computations fit loosely in the scope of one of these libraries.

8.1.5.2 Kokkos

Kokkos is a library designed for HPC clusters. It supports many backends (CUDA, HIP, SYCL, HPX, OpenMP and C++ threads). It inherits of all of the advantages listed for its backends, and is additionally oriented towards scalability across multi-node clusters.

This is very desirable to ensure scalability to huge computing platforms, such as what will be the Central Signal Processor (CSP) and Science Data Processor (SDP) of SKA.

Moreover, it has the same array feature than CuTe, (`View` vs `Tensor`), and is likely more popular and promising. However, it encourages a complete switch from the CUDA language, even if interoperability is available.

8.2 Multi-GPU, multi-node systems

At the scale considered in this thesis, single-GPU computing was generally enough to perform all computations. However, looking at cyberphysical systems of the next decade such as SKA or next-gen Thales radars, single-GPU will not be enough.

Data rates larger by one order of magnitude than considered in this work will have to be processed in real-time. Given the state of current hardware, this will have to be done on multi-GPU systems, and maybe even multi-node systems in the most extreme configurations.

Using distributed memory systems is the standard for HPC applications. Many frameworks exist to manage clusters and communicate between nodes. Some CUDA Toolkit libraries already feature multi-GPU execution capabilities (cuFFT, cuBLAS). Cutting-edge software even makes it possible to execute CUDA kernels designed for single-GPU execution on a multi-GPU configuration[26].

Once again, a tradeoff has to be addressed between dependency to Nvidia, development time and execution performance.

Fortunately, in the algorithms studied in the context of this work, most implementations can be distributed trivially, at least to some extent: identification of different streams / beams / tiles / has already been done during the parallelization process, to provide efficient CUDA implementations. This provides a good starting point on how to partition data, but not how to transmit it efficiently.

8.3 COSMIC framework

COSMIC[17] is a framework under development at LESIA - Observatoire de Paris, with the goal to implement features commonly required in real-time computing pipelines, such as composability, live reconfiguration, synchronization, telemetry and user interface.

Notably, each block of a pipeline is encapsulated in a so-called Business Unit (BU), an autonomous block connected to others in a pipeline through inputs and outputs, commonly resident in a shared GPU memory zone. This makes it possible to compose a pipeline in an abstract way, enabling live replacement of some BUs by other variants in order to optimize energy consumption, computation time, or adapt to more difficult observing conditions.

The user interface mainly consists of an automatic Python bindings generation. A pipeline is described using the Python programming language, and then controlled from the Python prompt, in order to handle BUs or access data from shared memories. Moreover, a Terminal User Interface (TUI) is made available, `processCTRL`, and makes it possible to manually launch, pause or step BUs respective main loops, as well as check the iteration number.

However, because a pipeline is launched from Python, some debugging tools such as `cuda-gdb` or `compute-sanitizer` are harder to use. Because of this, encapsulating the various pipeline components presented in this section is still an ongoing work, even though our libraries have been designed to facilitate this step.

Conclusion

In this part, we presented how we applied GPU computing techniques to the two main domains of interest of this thesis, radar systems and radioastronomy.

Notable contributions are a multiple very high performance GPU implementations of the 1D convolution specialized for complex inputs, ubiquitous in signal processing applications. The variety of implementations makes it possible to cover a large design space, while accommodating for the limitations of CUDA and its libraries. These implementations were used in order to provide a prototype of Massively Multi-Hypotheses Radar SP, a concept enabling efficient GPU usage in the context of radar signal processing, while increasing both the gain and the resolution of these processing chains. This prototype is already under use in order to design upcoming radar systems, and will be continuously improved as new optimizations are looked into.

The other major contribution of this work, in the context of radioastronomy, is an end-to-end FRB detection pipeline for NenuFAR. This pipeline is deployed on a machine on site in Nançay, and has already been validated by detecting three pulsars, sharing similar features with FRBs. This pipeline is launched on experiments opened to observation time sharing, and is fully automated, potentially leading to the first FRB detections on NenuFAR in the near future. Results from this pipeline still need verification by an astronomer in order to separate true from false positives, but the next iterations may be able to increase detection confidence. Moreover, we provide this code as open-source, in an effort to allow a reuse of its primitives, that could provide a technical solution to FRB detection on the upcoming giant telescope SKA. Many preprocessing and data processing methods could also be repurposed, being rather generic (median filtering, peak detections, …).

Finally, we discuss the connection between radar systems Signal Processing and FRB detection, that are closer than expected and could inspire one another, in order to improve detection confidence.

During this thesis, we confirmed that carefully selecting the algorithms is key in obtaining the best possible performance, especially on GPU. Indeed, the blind porting of a CPU-based application did not provide the expected performance, because of the lack of parallelism of the documented algorithms, and the bad performance of some operations on GPU, especially the presence of heavy branching.

By starting the implementation by focusing on high-level functionality, we were able to design the MMHRSP, a very good match to GPU execution, outperforming corresponding CPU implementations by two orders of magnitude.

Together with bringing higher performance density, GPUs generally deliver more FLOP/W and FLOP/$, and help reducing energy consumption of cyberphysical systems.

However, we highlight an important issue of the CUDA ecosystem, composed of a programming language but also many high quality libraries: as of writing, Nvidia supports this language only for Nvidia GPUs, effectively creating a dependency to this hardware vendor. This can cause problems for system maintainability, but also for sovereignty, a topic especially important in the context of radar systems.

Although alternatives exist, they are still behind compared to CUDA in terms of support, performance, libraries and overall adoption, even in the context of scientific computing and research.

Because of this, the choice between CUDA and its alternatives is difficult, as potential long-term benefits from alternatives to CUDA require a higher investment in terms of development time today, and may never become actually beneficial. Nevertheless, we believe that one of the

responsibility of academic research is to adopt, develop and enhance this kind of portable and open approach, and we will look actively into using alternatives to CUDA in the future.

Conclusion

During the course of this thesis, we successfully adopted two cutting-edge technologies, respectively related to data acquisition (userland networking, DPDK, GPUDirect) and data processing (GPU computing), making it possible to overcome the current performance limitations encountered in cyberphysical systems. This adoption was motivated by different applications, where these technologies were successfully used, notably for FRB real-time detection on the NenuFAR telescope, and primary radar data acquisition and processing. Additionally, our data acquisition system was successfully applied in the context of MICADO, an upcoming AO instrument for the ELT, and is being assessed for an upgrade of BIGCAT, the correlator of the ATCA telescope. Moreover, GPU computing was successfully used in order to port an existing Secondary Surveillance Radar SP from CPU to GPU.

The data acquisition system developed during this work exhibits very high-performance under multiple aspects: maximum bandwidth (close to 100 Gbit/s on 100 GbE hardware), latency ($\sim$4 µs per packet reception and processing, depending on the protocol complexity), or energy consumption (maximum load <2 % of one CPU core, and 5 % of the GPU).

This is made possible by using the DPDK library, and the associated GPUDirect capabilities, enabling NIC to GPU DMA, as well as careful PCIe usage based on increasing data locality in an effort to remove unnecessary PCIe transactions, batching, and careful GPU packet processing design based on persistent kernels.

However, this system still lacks generalization, and significant efforts are still needed to adapt it to a new application. Moreover, it bears a strong dependency to Nvidia hardware, that may need to be removed in order to be applicable at a large scale.

Additionally, we developed GPU computing pipelines with strong scientific and industrial impact.

The first such pipeline is the Massively Multi-Hypotheses Radar SP, a new architecture for radar systems, based on a great scale-up of the current architecture, enabled by GPUs, making it possible to process bigger input data rates while respecting real-time and improving gain throughout the SP chain. This makes it possible to detect fainter targets, with overall better resolution, and will provide a solution to detect two especially challenging classes of objects for the current airspace surveillance: drones, and hypersonic aircraft.

This new architecture made it possible to focus optimization efforts on one main operation, the one-dimensional convolution, a fundamental operation in signal processing, surprisingly lacking widely available optimized implementation for GPUs.

We provide multiple implementations of this fundamental operation featuring different complexities, starting from the quadratic complexity of the direct algorithm, and reducing to the quasilinear complexity of the FFT-based convolution, further optimized by implementing the overlap-save algorithm. A performance study of these implementations shows that the overlap-save method has much lower complexity, and can outperform a naive GPU implementation by up to a factor 100. It is the most appropriate solution for medium filters, from a few tens of elements and up to a few thousands, depending on the exact Compute Capability of the GPU used, as a result of a limitation in the library used to compute FFTs, cuFFTDx. Above this limit, a fallback to higher complexity algorithms is currently required, although other workarounds are envisioned.

Additionally, we developed execution time models, estimating computation time as a function of the input signal and filter dimensions. This can be used as a heuristic to find an optimal number of hypotheses, under the constraint of respecting real-time performance.

The second GPU computing pipeline developed in the context of this thesis was designed for FRB detection, with the goal to perform a continuous survey for FRBs on the NenuFAR telescope, effectively increasing tenfold the time dedicated to this task, and multiplying by the same factor the chances of detection of such an event in the low observational frequencies of NenuFAR. Moreover, this pipeline is able to work at a more precise time and frequency resolution than previous FRB search programs on NenuFAR by removing the need to store multiple hours worth of signal, opening the way to obtaining better SNR, and possibly the detection of fainter FRBs.

Such a pipeline was developed and deployed at Nançay, and has already been validated on 3 known pulsars, astronomical objects sharing similar characteristics with FRBs. It is currently able to detect FRB candidates, but is still subject to RFIs, and in consequence frequently produces

false positives. Future iterations of this pipeline will aim at increasing confidence in the results of this pipeline.

Again, execution time models were derived, showing very low usage of the GPU on standard configuration, and up to 40 % usage on the most extreme configuration. This is very encouraging regarding the scalability of this pipeline on even more extreme cyberphysical systems, such as SKA.

Overall, we showed that DPDK-based data acquisition coupled with GPU computing are viable options for computing platforms in current and upcoming extreme cyberphysical systems. However, care must be taken in order to avoid building a too strong dependency on Nvidia hardware, which poses new maintainability and sovereignty issues. Consequently, future major directions for this work include a study of alternatives to Nvidia hardware, and the compatibility of our proposed solution with hardware from other vendors, as well as a better generalization of our data acquisition system, enabling easier reuse of this cutting-edge approach.

Code availability

In favor of open science, and with the goal to promote both our high performance data acquisition system and our FRB detection primitives, we provide the code of our FRB detection pipeline for NenuFAR under LGPL: `https://gitlab.obspm.fr/jplante/NenuFRB-rt`.

Bibliography

[1] Rasim Alguliyev, Yadigar Imamverdiyev, and Lyudmila Sukhostat. "Cyber-physical systems and their security issues." In: *Computers in Industry* 100 (2018), pp. 212–223. ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2018.04.017. URL: https://www.sciencedirect.com/science/article/pii/S0166361517304244.

[2] M. Amiri et al. "The CHIME Fast Radio Burst Project: System Overview." In: *The Astrophysical Journal* 863.1 (2018), p. 48. DOI: 10.3847/1538-4357/aad188.

[3] Internet Assigned Number Authority, ed. *Internet Protocol Version 4 (IPv4) ParametersProtocol Numbers*. May 2018. URL: https://www.iana.org/assignments/ip-parameters/ip-parameters.xhtml#ip-parameters-1.

[4] Internet Assigned Number Authority, ed. *Protocol Numbers*. 2023. URL: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

[5] Toru Baji. "Evolution of the GPU Device widely used in AI and Massive Parallel Processing." In: *2018 IEEE 2nd Electron Devices Technology and Manufacturing Conference (EDTM)*. 2018, pp. 7–9. DOI: 10.1109/EDTM.2018.8421507.

[6] Ana Balevic. "Parallel Variable-Length Encoding on GPGPUs." In: *Euro-Par 2009 – Parallel Processing Workshops*. Ed. by Hai-Xiang Lin et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 26–35. ISBN: 978-3-642-14122-5.

[7] David C Brock. *Understanding Moore's law: four decades of innovation*. Chemical Heritage Foundation, 2006.

[8] P. Chris Broekema et al. "Cobalt: A GPU-based correlator and beamformer for LOFAR." In: *Astronomy and Computing* 23 (2018), pp. 180–192. ISSN: 2213-1337. DOI: https://doi.org/10.1016/j.ascom.2018.04.006. URL: https://www.sciencedirect.com/science/article/pii/S2213133717301439.

[9] Kendall Brown, Hugh Coleman, and W Steele. "Estimating uncertainty intervals for linear regression." In: *33rd Aerospace Sciences Meeting and Exhibit*. 1995, p. 796.

[10] Gareth Mitchell Callanan. "A GPU based X-Engine for the MeerKAT Radio Telescope." MA thesis. University of Cape Town, 2020.

[11] Daniel M. Coleman and Daniel R. Feldman. "Porting Existing Radiation Code for GPU Acceleration." In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 6.6 (2013), pp. 2486–2491. DOI: 10.1109/JSTARS.2013.2247379.

[12] Wei Dai and Feng Wang. "Study on processing performance of a dpdk and gpu combined pulsar data reduction system." In: *International Journal of Mechatronics and Applied Mechanics* 6 (2019), pp. 213–224.

[13] Toni Delovski et al. "ADS-B over Satellite The world's first ADS-B receiver in Space." In: (2014).

[14] DPDK, ed. *Overview of Networking Drivers*. 2023.

[15] Robert W Duffner. *The adaptive optics revolution*. Albuquerque, NM: University of New Mexico Press, June 2009.

[16] Paul Emmerich et al. "User Space Network Drivers." In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2019, pp. 1–12. DOI: 10.1109/ANCS.2019.8901894.

[17] F. Ferreira et al. “Hard real-time core software of the AO RTC COSMIC platform: architecture and performance.” In: *Adaptive Optics Systems VII*. Ed. by Laura Schreiber, Dirk Schmidt, and Elise Vernet. Vol. 11448. International Society for Optics and Photonics. SPIE, 2020, pp. 239–254. doi: 10.1117/12.2561244.

[18] R. J. Galejs and C. E. Muehe. *Beam and filter straddle losses in an ESA search radar*. Mar. 1992.

[19] D. C. Good et al. “First Discovery of New Pulsars and RRATs with CHIME/FRB.” In: *The Astrophysical Journal* 922.1 (Nov. 2021), p. 43. doi: 10.3847/1538-4357/ac1da6. url: https://dx.doi.org/10.3847/1538-4357/ac1da6.

[20] D. Gratadour et al. “MAVIS real-time control system: a high-end implementation of the COSMIC platform.” In: *Adaptive Optics Systems VII*. Ed. by Laura Schreiber, Dirk Schmidt, and Elise Vernet. Vol. 11448. International Society for Optics and Photonics. SPIE, 2020, p. 114482M. doi: 10.1117/12.2562082. url: https://doi.org/10.1117/12.2562082.

[21] Richard Hughes-Jones, Steve Parsley, and Ralph Spencer. “High data rate transmission in high resolution radio astronomy—vlbiGRID.” In: *Future Generation Computer Systems* 19.6 (2003). 3rd biennial International Grid applications-driven testbed event, Amsterdam, The Netherlands, 23-26 September 2002, pp. 883–896. issn: 0167-739X. doi: https://doi.org/10.1016/S0167-739X(03)00068-2. url: https://www.sciencedirect.com/science/article/pii/S0167739X03000682.

[22] ICAO, ed. *Aeronautical Telecommunications*. 5th ed. 2014. isbn: 978-92-9249-537-4.

[23] ICAO, ed. *Manual on Mode S Specific Services*. 2nd ed. 2004. isbn: 92-9194-407-6.

[24] “IEEE Standard for Ethernet.” In: *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)* (2018), pp. 1–5600. doi: 10.1109/IEEESTD.2018.8457469.

[25] *Internet Protocol*. Request for Comments 791. RFC Editor, Sept. 1981. 51 pp. doi: 10.17487/RFC0791. url: https://www.rfc-editor.org/info/rfc791.

[26] Jaehoon Jung et al. “SnuRHAC: A Runtime for Heterogeneous Accelerator Clusters with CUDA Unified Memory.” In: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’21. Virtual Event, Sweden: Association for Computing Machinery, 2021, pp. 107–120. isbn: 9781450382175. doi: 10.1145/3431379.3460647. url: https://doi.org/10.1145/3431379.3460647.

[27] Nick Karcher et al. “Versatile Configuration and Control Framework for Real-Time Data Acquisition Systems.” In: *IEEE Transactions on Nuclear Science* 68.8 (2021), pp. 1899–1906.

[28] Siddhartha Kumar Khaitan and James D. McCalley. “Design Techniques and Applications of Cyberphysical Systems: A Survey.” In: *IEEE Systems Journal* 9.2 (2015), pp. 350–365. doi: 10.1109/JSYST.2014.2322503.

[29] Volodymyr V. Kindratenko et al. “GPU clusters for high-performance computing.” In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009, pp. 1–8. doi: 10.1109/CLUSTR.2009.5289128.

[30] Ezra Kissel and Martin Swany. “Evaluating High Performance Data Transfer with RDMA-based Protocols in Wide-Area Networks.” In: *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. 2012, pp. 802–811. doi: 10.1109/HPCC.2012.113.

[31] J. R. Klauder et al. “The theory and design of chirp radars.” In: *The Bell System Technical Journal* 39.4 (1960), pp. 745–808. doi: 10.1002/j.1538-7305.1960.tb03942.x.

[32] Donald E. Knuth. “Structured Programming with Go to Statements.” In: *ACM Comput. Surv.* 6.4 (Dec. 1974), pp. 261–301. issn: 0360-0300. doi: 10.1145/356635.356640. url: https://doi.org/10.1145/356635.356640.

[33] Donald Ervin Knuth. *The Art of Computer Programming*. Addison-Wesley, 1969. isbn: 0-201-03801-3.

[34] Michail-Alexandros Kourtis et al. “Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration.” In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. 2015, pp. 74–78. DOI: 10.1109/NFV-SDN.2015.7387409.

[35] Jim St. Leger. “Myth-busting DPDK in 2020.” In: *AvidThink* (2020).

[36] Breno Henrique Leitao. “Tuning 10Gb network cards on Linux.” In: *Proceedings of the 2009 Linux Symposium*. Citeseer. 2009, pp. 169–185.

[37] Darryl D Lin and Sachin S Talathi. “Overcoming challenges in fixed point training of deep convolutional networks.” In: *arXiv preprint arXiv:1607.02241* (2016).

[38] Marco Lombardo et al. “Adaptive Optics Technology for High-Resolution Retinal Imaging.” In: *Sensors* 13.1 (2013), pp. 334–366. ISSN: 1424-8220. DOI: 10.3390/s130100334. URL: https://www.mdpi.com/1424-8220/13/1/334.

[39] D. R. Lorimer et al. “A Bright Millisecond Radio Burst of Extragalactic Origin.” In: *Science* 318.5851 (2007), pp. 777–780. DOI: 10.1126/science.1147532. eprint: https://www.science.org/doi/pdf/10.1126/science.1147532. URL: https://www.science.org/doi/abs/10.1126/science.1147532.

[40] S. Makhathini. “Advanced radio interferometric simulation and data reduction techniques.” Available via http://hdl.handle.net/10962/57348. PhD thesis. Drosty Rd, Grahamstown, 6139, Eastern Cape, South Africa: Rhodes University, Apr. 2018.

[41] Enrico Marchetti and Marcos Suarez-Valles. *Wavefront Sensor Cameras MUDPI Data Packet Description*. Ed. by ESO. Version 1.

[42] Vivien Marx. “Microscopy: hello, adaptive optics.” In: *Nature Methods* 14.12 (Dec. 2017), pp. 1133–1136. ISSN: 1548-7105. DOI: 10.1038/nmeth.4508. URL: https://doi.org/10.1038/nmeth.4508.

[43] Víctor Mayoral-Vilches et al. “Robotcore: An open architecture for hardware acceleration in ros 2.” In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2022, pp. 9692–9699.

[44] Raimund Meyer, Richard Reng, and Karl Schwarz. “Convolution algorithms on DSP processors.” In: *Acoustics, Speech, and Signal Processing, IEEE International Conference on*. IEEE Computer Society. 1991, pp. 2193–2194.

[45] Elias Molina and Eduardo Jacob. “Software-defined networking in cyber-physical systems: A survey.” In: *Computers & Electrical Engineering* 66 (2018), pp. 407–419. ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2017.05.013. URL: https://www.sciencedirect.com/science/article/pii/S0045790617313368.

[46] Rolf Neugebauer et al. “Understanding PCIe Performance for End Host Networking.” In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’18. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 327–341. ISBN: 9781450355674. DOI: 10.1145/3230543.3230560. URL: https://doi.org/10.1145/3230543.3230560.

[47] Jordan Novet. “AMD’s stock pops after Microsoft tech chief touts chipmaker’s AI products.” In: *CNBC* (Sept. 2023).

[48] Nvidia, ed. *CUDA C++ Programming Guide*. 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[49] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. Prentice-Hall, 1999. ISBN: 0137549202.

[50] Michael L Overton. *Numerical computing with IEEE floating point arithmetic*. SIAM, 2001.

[51] Thomas Pany et al. “The multi-sensor navigation analysis tool (MuSNAT)–architecture, LiDAR, GPU/CPU GNSS signal processing.” In: *Proceedings of the 32nd International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2019)*. 2019, pp. 4087–4115.

[52] Christian Patauner et al. “FPGA based microserver for high performance real-time computing in Adaptive Optics.” In: *Proceedings of the Adaptive Optics for Extremely Large Telescopes 5* (2017). doi: 10.26698/ao4elt5.0121.

[53] Denis Perret et al. “Bridging FPGA and GPU technologies for AO real-time control.” In: *Adaptive Optics Systems V*. Vol. 9909. SPIE. 2016, pp. 1364–1374.

[54] Hermann Rohling. “Ordered statistic CFAR technique - an overview.” In: *2011 12th International Radar Symposium (IRS)*. 2011, pp. 631–638.

[55] Alessio Sclocco et al. “Radio Astronomy Beam Forming on Many-Core Architectures.” In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 2012, pp. 1105–1116. doi: 10.1109/IPDPS.2012.102.

[56] Merrill I Skolnik. *Introduction to Radar Systems*. 3rd ed. New York, NY: McGraw-Hill Professional, Dec. 2002.

[57] G. Turin. “An introduction to matched filters.” In: *IRE Transactions on Information Theory* 6.3 (1960), pp. 311–329. doi: 10.1109/TIT.1960.1057571.

[58] Robert K. Tyson. *Principles of adaptive optics*. 1991.

[59] Manuel Ujaldon and Umit V. Catalyurek. “High-performance signal processing on emerging many-core architectures using cuda.” In: *2009 IEEE International Conference on Multimedia and Expo*. 2009, pp. 1825–1828. doi: 10.1109/ICME.2009.5202878.

[60] *User Datagram Protocol*. Request for Comments 768. RFC Editor, Aug. 1980. 3 pp. doi: 10.17487/RFC0768. url: https://www.rfc-editor.org/info/rfc768.

[61] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. “MIDeA: a multi-parallel intrusion detection architecture.” In: *Proceedings of the 18th ACM conference on Computer and communications security*. 2011, pp. 297–308.

[62] M Mitchell Waldrop. “More than moore.” In: *Nature* 530.7589 (2016), pp. 144–148.

[63] Naigang Wang et al. “Training deep neural networks with 8-bit floating point numbers.” In: *Advances in neural information processing systems* 31 (2018).

[64] Stuart Weston, Timothy Natusch, and Sergei Gulyaev. “Radio Astronomy data transfer using KAREN network.” In: *2011 XXXth URSI General Assembly and Scientific Symposium*. 2011, pp. 1–4. doi: 10.1109/URSIGASS.2011.6051235.

[65] Wayne Wolf. “Cyber-physical systems.” In: *Computer* 42.03 (2009), pp. 88–89.

[66] Philippe Zarka et al. “The low-frequency radio telescope NenuFAR.” In: *GASS*. URSI. Sept. 2020.

Appendix A

Papers

In this section, we reproduce two papers written during this thesis. The first one was submitted to the proceedings of the ADASS XXXI conference, that took place in Cape Town in 2021, but was never published. The second paper was published in SPIE Proceedings Vol. 12189: Software and Cyberinfrastructure for Astronomy VII.

A Novel FRB Detection Pipeline For NenuFAR

Julien Plante ,[1] Damien Gratadour,[1] Louis Bondonneau,[1] and Cédric Viou[2]

[1]*LESIA, Observatoire de Paris, Université PSL, CNRS, Sorbonne Université, Université de Paris, Meudon, Hauts-de-Seine, France;* julien.plante@obspm.fr

[2]*Station de Radioastronomie de Nançay, Observatoire de Paris, PSL Research University, CNRS, Université d'Orléans, Nançay, Cher, France*

Abstract. NenuFAR is a very large low-frequency radiotelescope, designed to observe the sky at frequencies ranging from 10 to 85 MHz. One of the main science case is the study of pulsars and Fast Radio Bursts (FRBs), and this program already has detected hundreds of these objects and events. However, the current detection pipeline is not able to perform a 24/7 survey of the sky. This is not an issue for the observation of recurring events such as pulsars emissions, but limits the number of FRB detections, and discovery opportunities. Here, we propose a novel real-time FRB detection system for NenuFAR, including a custom hardware platform and a software solution, designed to detect transient events in real-time and trigger signal storage on event detection to reduce memory footprint. This experiment on NenuFAR has also been designed as a pathfinder for more efficient transient events detection on SKA, with scalability as one of the core specifications. In this paper, we present the design of the experiment, detail the underlying hardware and software technologies and discuss initial results from a benchmarking campaign.

1. Introduction

Multiple Fast Radio Burst (FRB) detection pipelines are available nowadays (Petroff et al. 2019). Because of the low observation frequency of the NenuFAR telescope (10-85MHz) (Zarka et al. 2020), the dispersive delays of FRBs are up to a thousand times greater than for usual radiotelescopes (with typical observations centered around 1GHz). This makes existing pipelines perform poorly on NenuFAR data (increased memory footprint and latency, reduced Dispersion Measure (DM) search capabilities).

To address this limitation, we propose a pipeline tailored for low-frequency, real-time FRB detection.

2. Experiment Description

The pipeline currently under development is summarized on Figure 1, while more in-depth explanations are given in this section.

NenuFAR data is multicasted to experiments using UDP packets. To maximize the bandwidth and minimize memory footprint, we use DPDK + GPUDirect. This cutting-edge technology introduced by Nvidia enables direct transfer from the Network Interface Controller (NIC) to the GPU, without the need for a bounce buffer on CPU.

Incoming data is distributed on GPUs by frames of a few seconds in a round-robin fashion. Using multiple GPUs increases the time available for one iteration, making more complex operations possible.

Radio Frequency Interference (RFI) mitigation is performed after a simple preprocessing (time integration + power law), to identify and remove signal parts contaminated by undesired radio signals such as human emissions. This stage is not yet implemented in our pipeline.

Dedispersion is then applied to compensate the dispersion effect caused by the Interstellar Medium (ISM). We based our work on the Dispersion Measure Transform (DMT), with a custom strategy to address the detection of highly dispersed (> 10 s) bursts. Using piecewise integration, we are able to compute the same DMT by iterating on shorter time frames, while being independent of the underlying DMT algorithm. Current implementation uses the Bruteforce DMT algorithm (Barsdell et al. 2012).

The final stage detects energy peaks in the DMT. We propose an image processing approach to take advantage of highly optimized GPU libraries for image processing, such as Nvidia Performance Primitives (NPP). The current implementation is very limited, simply based on a global maximum and threshold, but we are working on more robust algorithms.

FRB candidates will trigger an external ring buffer of several seconds-long based on memory-only storage. That will allow raw waveform data retrieval and their recording on HDD to bring only signals of interest to the final user.

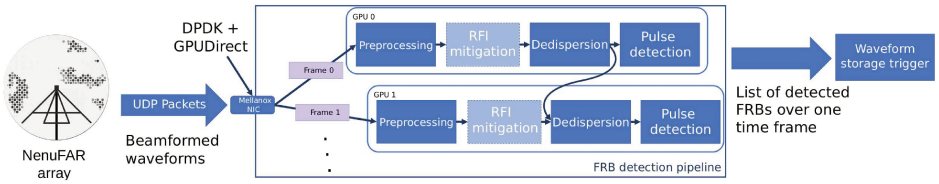


Figure 1. Pipeline architecture

This pipeline will be powered by the COSMIC framework (Ferreira et al. 2020) for real-time GPU computing, to provide stable performance, flexibility and a comprehensive user interface.

3. Initial Results

We compared the current performance of our pipeline to that of Heimdall, another GPU-accelerated FRB detection pipeline, and studied the new data acquisition approach.

3.1. DPDK + GPUDirect Benchmark

A benchmark has been realized to assess the performance of the DPDK + GPUDirect data acquisition method. Figure 2 show the results of a 1-minute benchmark of the latency, at 1.5 Gbit/s (the maximum bandwidth of NenuFAR for this experiment). These results were obtained with two Mellanox Connectx-5 interconnected with a 100 GbE fibre. The latency is measured as the delay from the first send to the full reception in GPU memory of 100kB chunks of data.

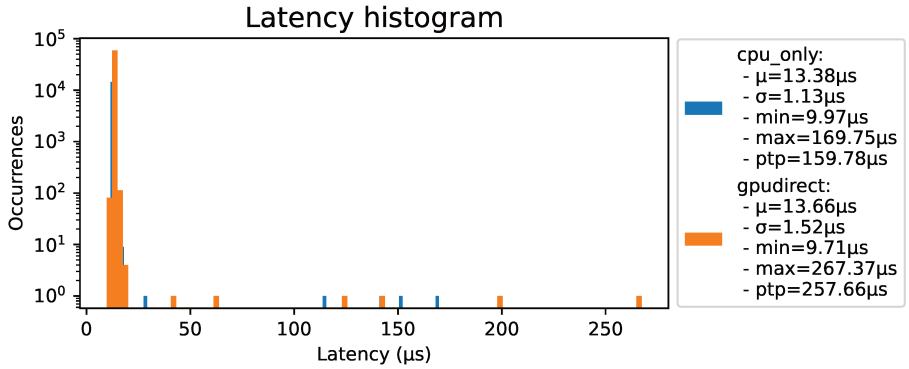


Figure 2. Latency over 1 minute at 1.5 Gbit/s

The mean latency in both NIC to CPU and NIC to GPU modes is about 13 µs, which will allow for a quick response of the detection pipeline. Some outliers are present, and will have to be dealt with to use this technology in a strong real-time environment. We believe this can be solved by tuning the system configuration (CPU shielding, real-time kernel, ...).

Moreover, we found that this result is strongly dependent on the PCIe topology of the experiment. With the NIC on the same PCIe bridge than the GPU, bandwidths up to 80 Gbit/s were measured. However, with a different topology, bandwidths down to 2 Gbit/s were noticed, with much higher latencies (hundreds of µs).

3.2. Computations Benchmark

To compare with Heimdall, we used NenuFAR-like data: 200 kHz sampling frequency (with time integration over 16 samples), 2^{21} samples per frame, 256 DM. The benchmark was run on one Nvidia Quadro GV100 (no multi-GPU for fair comparison of the implementations).

	Heimdall	This pipeline
RFI mitigation	25 s	N/A
Dedispersion	3.3 ms	4.6 ms
Pulse detection	800 ms	6.8 ms

Table 1. Computation time comparison for each stage

Both Heimdall and our pipeline are already able to detect FRBs in real-time (2^{21} samples at 200 kHz is about 10.5s) using high-end CPU and GPU, and still have a great margin for their processing time compared to a single frame time. This gives us room for thorough RFI mitigation, or opens the door to using lower-end hardware.

Our current implementation of dedispersion is 40% slower than Heimdall's, but enables the detection of FRBs with a dispersive delay longer than the time frame duration. It also allows easy pipelining across multiple GPUs, thus increasing the available processing time with each added GPU. Still, our implementation is very naive, and speedups are expected with further optimizations.

Our implementation of pulse detection is more than 100x faster than Heimdall's, likely thanks to our 2D approach to pulse detection in the DMT space. Though this approach is promising, the current implementation is not robust to RFIs, and needs fine-tuning of a threshold.

4. Conclusion

We presented initial results concerning a new FRB detection pipeline tailored for use with NenuFAR. This system will make possible to scale down the memory requirement for FRB study, by detecting FRBs on-the-fly, and saving only the FRB-related signal on disk.

Using DPDK + GPUDirect, we are able to provide a low-latency, high-bandwidth, and rather standard data acquisition method. While this technology is currently implemented using Nvidia hardware, we believe this could be ported to other manufacturer's hardware. This data acquisition method is more than enough to fill the requirements in this application, but enables high scalability.

A new approach to dedispersion emerged from this work, especially useful for low-radio observation, by removing the need for overlap and making possible to detect pulses longer than the time frame studied. This approach easily enables multiple-GPUs parallelism, increasing the available time for processing one time frame.

This progress is promising for the deployment of an online robust FRB detection pipeline for the low-frequency radiotelescope NenuFAR, with a first prototype integration planned for the first half of 2022.

5. Perspectives

The current priority is to complete the pipeline by linking the DPDK + GPUDirect to the rest of the pipeline. Further important steps are to implement a state-of-the-art RFI mitigation tuned for NenuFAR, and more robust pulse detection algorithms to provide an end-to-end prototype ready for integration at Nançay.

Next iterations will also focus on optimizing the DMT implementation, either by optimizing memory accesses in the current Bruteforce DMT, or by using the Fast DMT (Zackay & Ofek 2017), an algorithm with reduced complexity for the DMT computation. The outliers in latency for the data acquisition method are also to be looked into.

References

Barsdell, B. R., Bailes, M., Barnes, D. G., & Fluke, C. J. 2012, MNRAS, 422, 379. 1201.5380

Ferreira, F., Sevin, A., Bernard, J., Guyon, O., Bertrou-Cantou, A., Raffard, J., Vidal, F., Gendron, E., & Gratadour, D. 2020, in Adaptive Optics Systems VII, edited by L. Schreiber, D. Schmidt, & E. Vernet, International Society for Optics and Photonics (SPIE), vol. 11448, 239

Petroff, E., Hessels, J. W. T., & Lorimer, D. R. 2019, The Astronomy and Astrophysics Review, 27, 4. URL https://doi.org/10.1007/s00159-019-0116-6

Zackay, B., & Ofek, E. O. 2017, The Astrophysical Journal, 835, 11

Zarka, P., Denis, L., Tagger, M., Girard, J., Coffre, A., Dumez-Viou, C., Taffoureau, C., Charrier, D., & other members of the NenuFAR-France team 2020, in GASS (URSI)

A high performance data acquisition on COTS hardware for astronomical instrumentation

Julien Plante[a], Damien Gratadour[a], Lionel Matias[b], Cédric Viou[c], and Elena Agostini[d]

[a]LESIA, Observatoire de Paris, Université PSL, CNRS, Sorbonne Université, Université de Paris, Meudon, Hauts-de-Seine, France
[b]Thales LAS France, Limours, Essonne, France
[c]Observatoire Radioastronomique de Nançay, Observatoire de Paris, PSL Research University, CNRS, Université d'Orléans, Nançay, Cher, France
[d]NVIDIA, Santa Clara, CA, USA

ABSTRACT

Data throughput in modern telescopes instrumentation have been steadily increasing over the last decade. The few gigabits per second range is now the lower bound, and bandwidths as high as tens of terabits per second are expected with the Square Kilometer Array. We present a new approach based on DPDK, and its support for GPUDirect recently introduced by Nvidia to perform DMA from Network Interface Controller (NIC) to GPU memory, to answer very high throughput data acquisition in astronomy.

Keywords: Data acquisition, GPU, DPDK, GPUDirect

INTRODUCTION

Telescopes, such as other cyber-physical systems, share a similar high-level architecture: information from the physical world is captured through sensors (e.g. cameras or antennas), sent to a processing unit, and an action is taken (e.g. storing an image or a spectrum, steering an antenna or activating an actuator).

Much progress has been made on the processing side, with systems increasingly hybrid, combining multicore CPUs, GPUs, and others to reach the required computing power. The advent of the CUDA programming language especially made for GPUs has become a first-class general purpose computing hardware, bringing together very high throughput and user-friendly programming and low cost maintainability.

Although the processing side has seen many actionable improvements, we cannot say so for the data acquisition side. Even though some major progress on the hardware side has to be acknowledged (10/40/100/200/... GbE, Infiniband), their acceptance and use is still limited. Some high level programming models, such as MPI, are already able to use these features at their full power, but little work has been made to fit the needs of general purpose data acquisition.

In this paper, we present a high performance data acquisition prototype and provide performance estimates. This prototype is based on Commercial-off-the-Shelf (COTS) hardware, targeted at astronomical instrumentation together with GPU computing. We will begin in section 1 with a more in-depth explanation of the issue of data acquisition, covering the mechanism and the current approaches. Then, we present the different applications that motivated this work in section 2, and that will be used as examples and test cases during the rest of the article. We describe the current prototype in section 3, and show its performance in section 4. Finally, we state further possible improvements to the current prototype in section 5, and conclude.

Further author information:
Julien Plante: E-mail: julien.plante@obspm.fr

1. DATA ACQUISITION IN CYBER-PHYSICAL SYSTEMS

1.1 Ubiquity of the UDP protocol

To transfer data between different parts of a system which processes a continuous stream of data, a commonly used approach is to rely on the UDP (User Datagram Protocol) network protocol, together with some application specific information. The UDP protocol is very lightweight, containing only a checksum as security mechanism. It does not guarantee delivery nor ordering, resulting in less overhead and simpler emitter/receiver, but requiring some security mechanisms to be implemented on top.

This simple design explains its ubiquity in cyber-physical and real-time systems, together with some historical factors. Let's now see how a data acquisition system can be built for this kind of protocol when using GPUs.

1.2 Naive approach and limitations

The naive approach to perform computations on streaming data on a platform hosting a number of CPUs and GPUs can be described as the following:

- Receive data through the Linux network stack (recv / recvmsg / recvmmsg)
- Process the application specific header on CPU, and store the payload in a ring buffer in RAM
- Copy a chunk of the ring buffer to GPU memory
- Process this chunk of memory on the GPU
- Take action

The issue with this design is two-fold:

- Each call to a Linux networking function requires a system call, which has a moderate to high impact on performance. The worst case is when this system call requires a context switch. A better case is when vDSO is used, but overall, it is hard to receive more than 10 Gbit/s per CPU core because of this.
- Data has to be copied twice across the PCIe (NIC $\rightarrow$ RAM, RAM $\rightarrow$ GPU), which induces additional unwanted latency.

1.3 Related work

RDMA is possible over an Infiniband network fabric, or over an Ethernet fabric using RoCE (RDMA over Converged Ethernet). These solutions are arguably the most widespread, and provide good overall performance. However, this is mostly used for node-to-node communication in HPC, as it requires a level of logic on the emitter side. In cyber-physical systems, data often comes from sensors, or very simple, high-throughput hardware. The level of control over this emitting side can be very low, and implementing RoCE was not conceivable for the applications motivating this work.

Sending packets directly from a network controller to the GPU memory has been achieved for Adaptive Optics (AO) using a custom FPGA board,[1] achieving line throughput and very low latency at the price of dedicated hardware and firmware, rather difficult to maintain. The motivation behind our work is to reiterate this result on Commercial-off-the-shelf (COTS) hardware, to reduce the development time, help maintenance and ease the development of future high-performance data acquisition systems.

Many projects have been initiated in order to overcome the performance bottleneck of the Linux networking stack, the most adopted solutions being the Data Plane Development Kit (DPDK)[2] and Programming Protocol-independent Packet Processors (P4).[3] Some projects in astronomy already use such frameworks as a data acquisition method, such as the CHIME detector[4] and YAO's 40-m telescope.[5]

However, this has only ever been used to replace calls to the Linux networking stack, still using a RAM-based ring buffer. NVIDIA recently introduced a new DPDK library named gpudev[6] to help GPU real-time packet

processing applications where CPU is in the critical path to coordinate the network card to receive packets in GPU memory (technology branded as "GPUDirect RDMA" by NVIDIA) and notifying a packet-processing CUDA kernel waiting on the GPU for a new set of packets. The main requirement is to maximise the zero-packet loss throughput at the lowest latency possible. The gpudev library, not only introduces GPUDirect RDMA in the DPDK ecosystem but also offers utilities like the so called "communication list" to provide a real-time direct information exchange tool to CPU and GPU. We wanted to apply this technique to astronomy in order to remove all the costly operations mentioned in subsection 1.2.

2. TARGETED APPLICATIONS

2.1 Adaptive Optics

Our work has been chosen as a solution for the acquisition of wavefront sensor (WFS) image frames in the context of MICADO's SCAO (Single-Conjugate Adaptive Optics),[7] one of the adaptive optics systems of the ELT. This application is the main driver for very low latency, as the round-trip time through the controller in this context must be below 300µs. Ingress bandwidth is of about 2 Gbit/s, and as such is not the main challenge.

This application is also the only one explored with burst emission: in SCAO, a WFS image frame is broken down into 1 leader packet, N payload packets and 1 trailer packet. The case addressed by this work relates to the ESO-ALICE WFS camera, where N=60. This burst emission mode can be compared to a continuous emission type, where every packet is a payload packet, without discretization of separate image frames.

Reception has already been validated for this use on a first prototype, and is being fully qualified. The processing part of the pipeline has already been implemented as modules in the COSMIC framework.[8]

2.2 Radioastronomy

The original exploration ground for this work is a FRB pre-detection pipeline for NenuFAR.[9] The goal of this project is to perform a continuous Fast Radio Burst (FRB) survey on NenuFAR, while only triggering storage of the full-resolution time-frequency data through candidate FRB pre-detection. This would make it possible to detect a larger number of new FRBs, without a huge memory footprint.

A collaboration has also started with CSIRO to use this work for the ongoing upgrade of ATCA-BIGCAT. This application is especially challenging because of the higher bandwidth need, from 60 Gbit/s to 120 Gbit/s depending on the final design choices. The multiple streams configuration of this setting is also interesting, as we rely on hardware packet filtering to distribute the whole data stream to multiple compute nodes.

2.3 Radar

The final application of this work currently in development relates to radar systems, through a collaboration with Thales LAS France. Radar systems at Thales share the same high-level architecture than radio-telescopes on the reception side, and can take advantage of a high performance data acquisition system. Bandwidth requirements range from 40 Gbit/s to 1.6 Tbit/s, with an overall timing budget of a few hundreds of microseconds per algorithm iteration, implying very low latency in this application as well.

A functioning prototype has been developed for this project, with which we have demonstrated the reception up to 100 Gbit/s. This maximum bandwidth is currently limited by the hardware used in this setup.

2.4 Reusability

This work is being implemented as a library, and offers an optional wrapper with the COSMIC framework.[8] Both solutions are being developed to help building high performance data acquisition for GPU-based computing nodes. While the library itself is not ready to be distributed yet, a first glance at the code for the NenuFAR application can be made in this repository: `https://gitlab.obspm.fr/jplante/NenuFRB`. The Acquisition folder implements the work exposed in this article.

The main difficulty in building this work into a reusable component is the wide variety of protocols. One idea would be to build upon P4[3] for the protocol description syntax, and to generate CUDA + DPDK code from there.

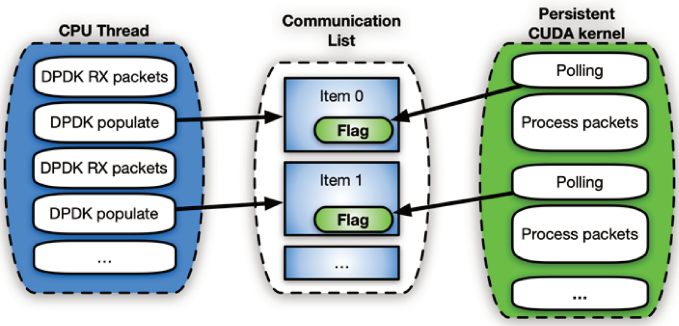


Figure 1. DPDK gpudev communication list mechanism to coordinate in real-time CPU receiving packet and notifying the GPU and GPU processing them. In this way, network and processing can be parallelized.

3. CURRENT PROTOTYPE

3.1 Design and limitations

As mentioned in subsection 1.3, by means of the DPDK gpudev library in order to reproduce the sequence in subsection 1.2 it's possible to:

- Initialise network card to use GPU memory for network packets creating a GPU memory DPDK mempool
- Pre-launch a CUDA kernel, waiting on GPU to receive a new set of packets to process (persistent or semi-persistent CUDA kernel)
- Create a communication list to coordinate CPU and GPU progresses
- Receive packets in GPU memory using DPDK `rte_eth_rx_burst` function

Once these items are in place the interaction between CPU and GPU can be described as in Figure 1:

- CPU thread keeps receiving packets in GPU memory and populate communication list items with packets info (number, memory addresses, etc..)
- CUDA packet processing kernel waits on the next item of the communication list to get CPU notification about new packets received in GPU memory
- Process the new subset of packets received in GPU memory as in Figure 2

Pros of this approach are many. Pre-launching a CUDA kernel waiting on the communication list items for a CPU notification avoids to launch a new CUDA kernel for each subset of receiving packets (less CPU work in critical path). Additionally, the whole Ethernet packet is received in GPU memory so protocol termination can be done directly within the CUDA kernel where packets' headers and payload can be analysed and real-time decisions can be made (as in Figure 2).

Moreover, packet processing is a GPU software approach so whatever type of analysis an application has to do (data placement, traffic analysis, etc..), it can be implemented regardless the subsystem (type of CPU, type of network card, etc..) without relying on specific network card offloads or CPU capabilities taking the advantage of the high degree of parallelism of GPUs being able to process tons of packets in parallel.

A first limitation of this design is that the CPU is still on the critical path. One core is required to poll the NIC for new ingress packets. Although this core is doing very little work, it is looping infinitely in order to reduce jitter as much as possible, resulting in a 100% CPU load. Similarily, this solution requires by design a subset of a GPU to perform packet processing. This contributes to the great flexibility and performance of this approach, but has to be taken into account as it could interfere the overall GPU performance.

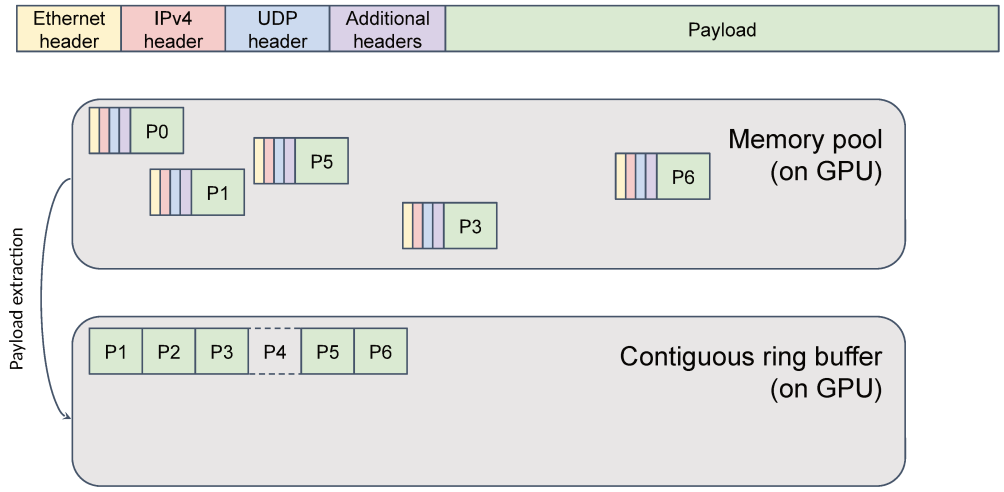


Figure 2. General representation of packet reconstruction. Packets arrive in a GPU memory pool managed by DPDK, which does not guarantee any ordering. We thus have to parse each incoming packet to extract the payload to a contiguous memory location.

A second limitation is that this technology has been introduced by Nvidia, and only been tested on Nvidia hardware (Mellanox NICs and Nvidia GPUs). While the concept of DMA is a standard PCIe feature, and should be usable on other hardware, this has not been verified yet.

Finally, a third limitation is that the user needs to write a GPU kernel in CUDA to process the application specific header on GPU. Each new protocol needs a part of rather low-level code to be written, which reduces portability. However, writing this packet processing kernel allows the user to have complete control about this packet processing operation, which enables the implementation of many features: complex networking protocols, preprocessing (time integration, power law, ...), signal transposition and so on.

4. BENCHMARKING

Two benchmarking machines have been used to verify the functional behaviour and measure the performance in terms of latency and throughput of our data acquisition method. The only major difference between the two machines is the version of PCIe bus. We observed that the PCIe gen 3 limits the maximum bandwidth at 80 Gbit/s, contrary to the 4th generation with which we reached 100 Gbit/s.

To facilitate latency measurement, experiments have been made with a physical loopback Ethernet link connecting the NICs of the machine handled by different CPU cores. This avoids clock synchronization issues, which would be predominant at our time scale ($< 100\mu s$). Experiments are planned to check the functional behaviour on two different machines, but will not be addressed in this article.

During our experiments on “moksha” we learnt about the importance of the PCIe topology when using this data acquisition technology. Going through multiple PCIe switches can decimate the achievable throughput, and much care must be taken to the PCIe topology because of this.

4.1 Packet processing kernel bandwidth

A preliminary experiment was to check whether the packet processing GPU kernel (see Figure 2) was able to keep up with the ingress throughput. We measured the throughput of this kernel, executed on one CUDA block and a varying number of CUDA threads per block, as well as a varying size of memory transactions (Figure 5).

Figure5 helps sizing the required GPU resources to run the persistent packet processing kernel. We observe a sweet spot for 512 threads per block, which can be explained by the high cost of memory transactions (hundreds of clock cycles per transaction). Using more threads allows the hardware to better hide these latencies, at the price of higher contention on the memory controller. Up to 512 threads per block, the latency hiding mechanism

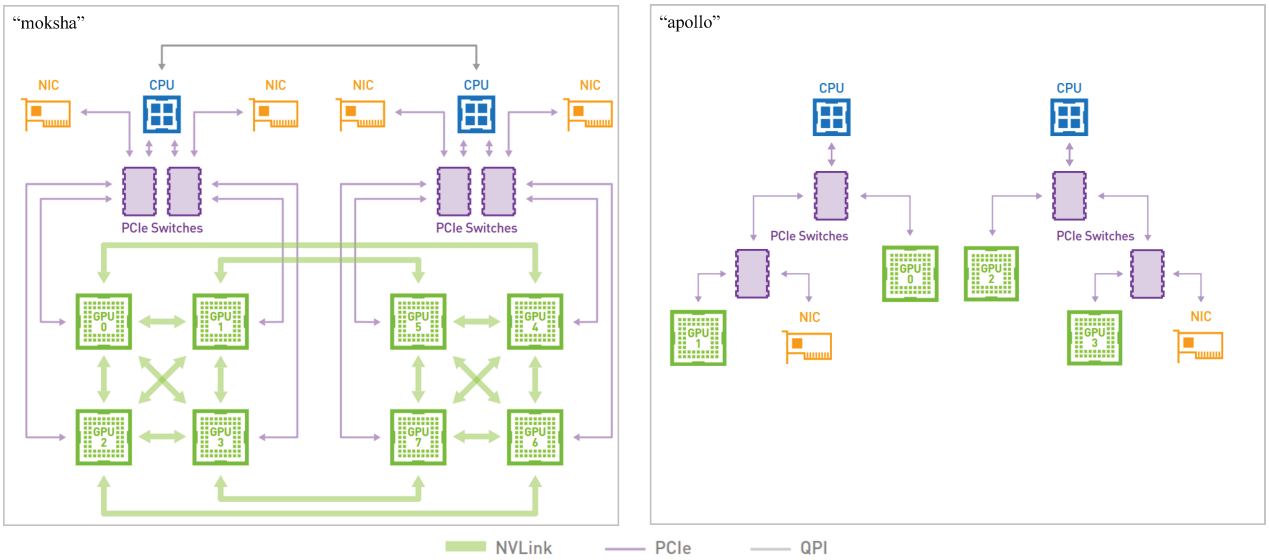


Figure 3. The two test machines, "moksha" and "apollo". Moksha features a PCIe Gen 3 bus, while apollo features a PCIe Gen 4 bus. On moksha, the leftmost NIC is connected to the rightmost NIC through a 100GbE Ethernet. On apollo, the two NICs are interconnected using a 100GbE Ethernet cable, that can be upgraded to a 200GbE Ethernet cable.

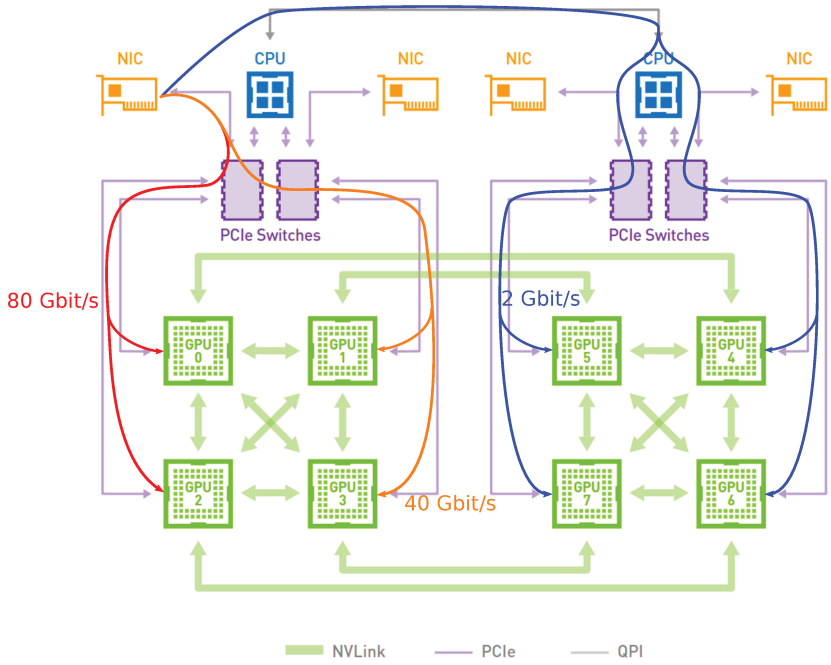


Figure 4. Mean bandwidth on PCIe generation 3 for DMA between a NIC and a GPU varying through different paths. The best bandwidth is achieved for devices that are on the same PCIe bridge.

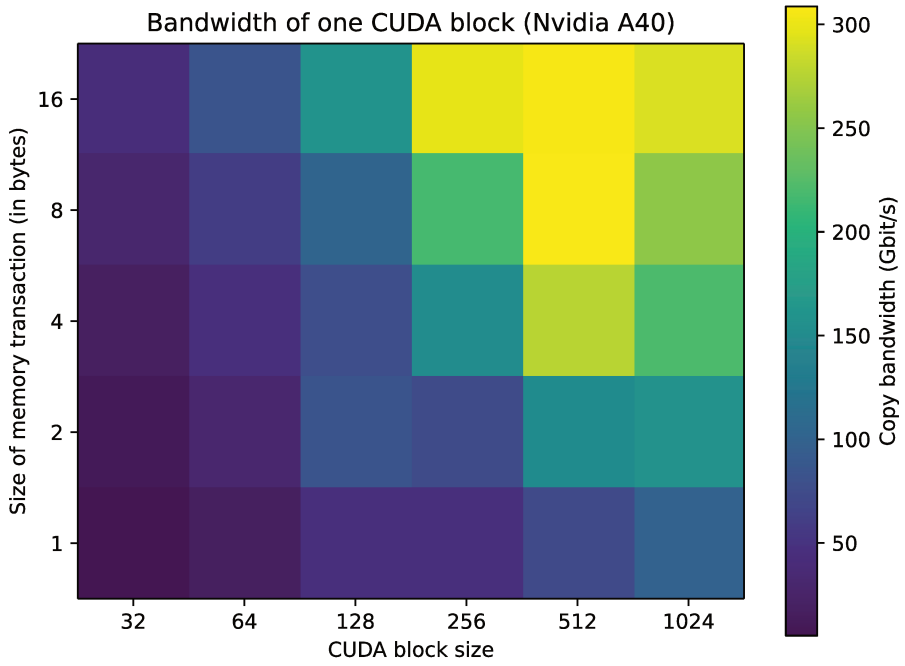


Figure 5. Measured copy bandwidth of one CUDA block on Nvidia A40. Memory transactions can be vectorized on GPU up to 16-byte transactions, as long as the memory accessed is aligned to the same boundary.

prevails, leading to an increased bandwidth. After this limit, the higher hardware contention becomes limiting, resulting in worse performance for 1024 threads per block.

We also confirm that vectorization using bigger memory transactions improve the copy bandwidth. Finally, we notice that most configurations can handle 100 Gbit/s, and many of them can sustain 200 Gbit/s. We deduce that a kernel of only one CUDA block is enough to perform packet processing. Using only one block will help keeping the performance impact of this approach limited.

4.2 Adaptive Optics experiment

Other experiments were focused on simulating the reception of the multiple real-life applications exposed in section 2. The Adaptive Optics application was tested on the "moksha" machine, with the leftmost NIC sending ALICE WFS-like packets, and the rightmost NIC receiving. Processing is done on GPU 4, which is on the same PCIe bridge as the receiving NIC.

The functional behaviour for this application is correct, with image frames being reconstructed as expected and no packet being lost. We measured the latency between the reception of the trailer packet and the end of the packet processing for each frame (Figure 6).

Two implementations are compared. A reference one acquiring packets with DPDK on CPU, and performing the packet processing on CPU before uploading data to the GPU afterwards using a cudaMemcpy. Basic optimizations were used to make a fair comparison (vectorization, pinned memory). The second implementation relies on DPDK with GPUDirect, as described in the other parts of this article. Both implementations are based on the COSMIC framework.

We observe a mean latency five times smaller with our prototype than with a more naive approach. However, we notice a relatively high jitter: even though the standard deviation of this recording is small, the peak-to-peak latency is very high with regular spikes of latency (4x mean latency). This is most likely due to the presence of the CPU on the critical path. While this result is acceptable and shows the feasibility of a COTS acquisition system, it could be improved and especially optimized for jitter, as discussed in 6.1.

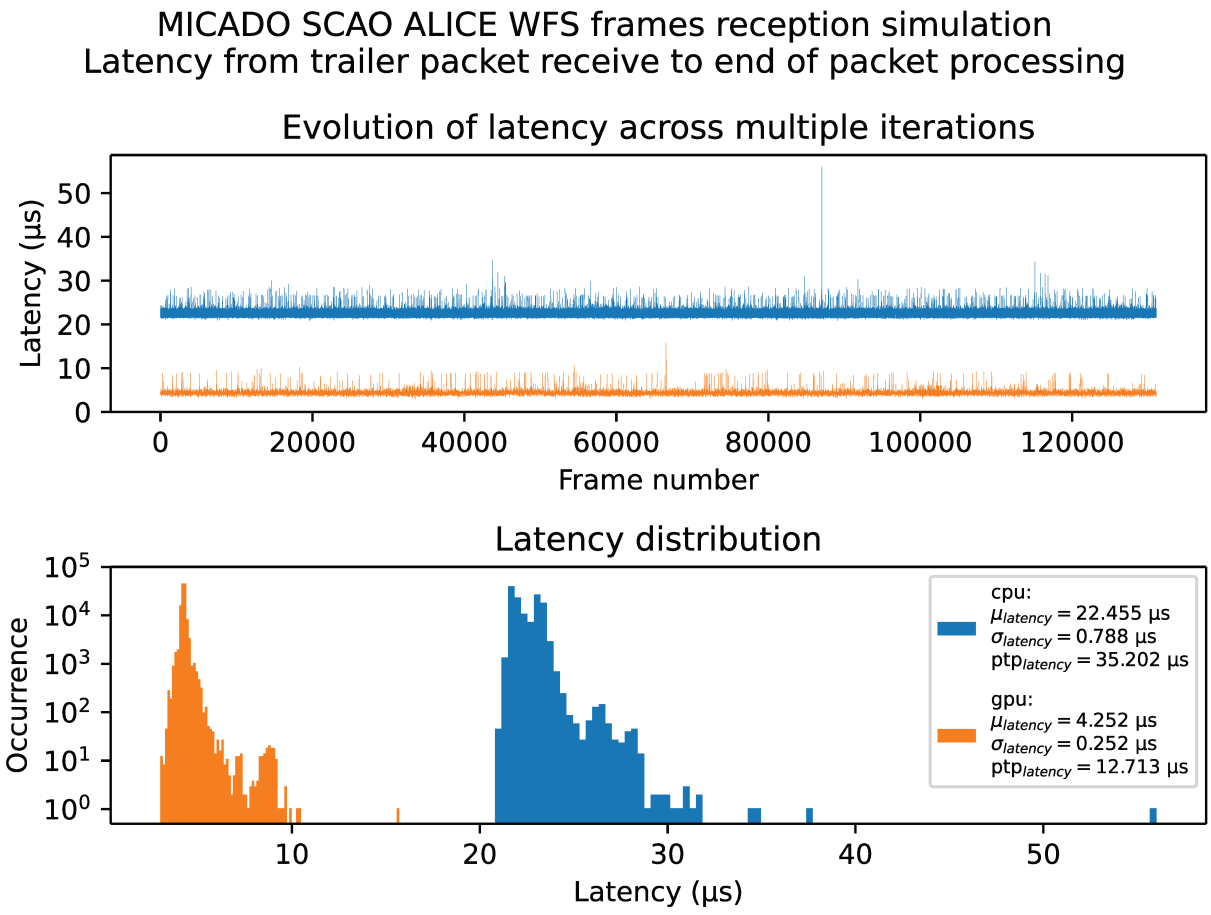


Figure 6. Latency measured for 2^{17} frame transfers

4.3 Other experiments

For other applications, the functional behaviour has been validated, up to 100 Gbit/s for radar, with no packet loss. This demonstration has been made using only one CPU core, and one CUDA persistent kernel of one CUDA block. Latency measurements have not been performed as of now, since the latency requirement is weaker than for adaptive optics.

For the NenuFAR application, the behaviour has been verified using packet captures of the actual network traffic of the telescope, stored under the PCAP format. This is a great step in the writing of the FRB predetection pipeline for NenuFAR,[9] and will allow us to focus on the processing part.

For radars, the behaviour has only been verified on a custom packet generator written with DPDK, because of the lower technlogy readiness level, but show very promising results. Indeed, these first results show that this acquisition method can be used to achieve line throughput on COTS hardware.

5. FUTURE WORK

5.1 GPU-driven DMA

One of the biggest limitation of the current design is that CPU is still on the critical path, since the packet reception is driven by the CPU. Previous work show that CPUs are the main source of jitter in real-time hybrid systems, even when using CPU shielding. Moreover, our design only requires the presence of the CPU as an orchestrator because of DPDK, it does not serve any other purpose currently.

One of the future goals would be to get rid of the CPU in the main loop of reception, and to drive the DMA transfer from the GPU itself. The expected gain of this shift would be to further reduce the latency and jitter observed, as well as freeing the CPU core running the host-side reception loop. Depending on the capabilities of such a GPU-driven DMA, it could also reduce the complexity of the data acquisition system, by reducing the race condition hazards and the number of required synchronizations.

5.2 OS-level optimizations

The “moksha” machine runs Ubuntu 20.04 LTS, together with the Linux kernel 5.4.0-97-generic. Further experiments will focus on CPU shielding and using a real-time kernel.

5.3 Security mechanisms

The current implementation feature little security mechanisms. Ingress data is stored in a ring buffer, without any meta information. This implies that old data is overwritten without any notice, and it is not possible to detect a missing packet.

A solution based on attaching an additional ring buffer containing this meta information to the data ring buffer is being investigated. The nature of this information could be made protocol-independent.

5.4 Targeted applications

We are gaining maturity in this data acquisition method with each of the various application we already have. All of them are only explored through simplistic packet emitters currently. The next step regarding these applications is to perform the data acquisition on a real stream of data, or on a better simulator depending on the cases. We especially plan to deploy the FRB predetection pipeline on NenuFAR by the end of the year, after completing the processing chain.

CONCLUSION

We presented a new high-performance data acquisition solution for hybrid systems based on COTS hardware, using DPDK and GPUDirect. The current prototype achieves low latency (30 µs) up to line bandwidth, and has been tested up to 100 Gbit/s with no packet drop. This work will further simplify the use of GPUs in computing nodes of cyber-physical systems, and will help reaching real-time performance on hybrid systems.

We are working to make this work usable for any protocol, looking at high-level protocol descriptions such as those proposed by P4.[3] As of now, the prototype developed can be adapted manually to any protocol, but requires a good understanding of the DPDK. Further iterations will aim at making this work easy to use as a replacement of more standard acquisition methods.

ACKNOWLEDGMENTS

Many thanks to Florian Ferreira from LESIA for his help on understanding the protocols used in MICADO-SCAO.

The PhD project of Julien Plante at Observatoire de Paris is sponsored through a grant from Paris Region PhD program (PRPHD 2020).

This work is sponsored through a grant from project 873120, a.k.a. Rising STARS, funded by European Commission under program H2020-EU.1.3.3 coordinated in H2020-MSCA-RISE-2019.

REFERENCES

[1] Patauner, C., Biasi, R., Andrighettoni, M., Angerer, G., Pescoller, D., Porta, F., and Gratadour, D., "Fpga based microserver for high performance real-time computing in adaptive optics," *Proceedings of the Adaptive Optics for Extremely Large Telescopes 5* (2017).

[2] Leger, J. S., "Myth-busting dpdk in 2020," *AvidThink* (2020).

[3] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., and Varghese, G. e. a., "P4," *ACM SIGCOMM Computer Communication Review* **44**(3), 87–95 (2014).

[4] Amiri, M., Bandura, K., Berger, P., Bhardwaj, M., Boyce, M. M., Boyle, P. J., Brar, C., Burhanpurkar, M., Chawla, P., and Chowdhury, J. e. a., "The chime fast radio burst project: System overview," *The Astrophysical Journal* **863**(1), 48 (2018).

[5] Dai, W. and Wang, F., "Study on processing performance of a dpdk and gpu combined pulsar data reduction system," *International Journal of Mechatronics and Applied Mechanics* **1**(6) (2019).

[6] Agostini, E., "Boosting inline packet processing using dpdk and gpudev with gpus | nvidia technical blog," (2022).

[7] Clénet, Y., Buey, T. M., Rousset, G., Cohen, M., Feautrier, P., Gendron, E., Hubert, Z., Chemla, F., Gratadour, D., and Baudoz, P. e. a., "Overview of the micado scao system," *SPIE Proceedings* (2014).

[8] Ferreira, F., Sevin, A., Bernard, J., Guyon, O., Bertrou-Cantou, A., Raffard, J., Vidal, F., Gendron, E., and Gratadour, D., "Hard real-time core software of the ao rtc cosmic platform: architecture and performance," *Adaptive Optics Systems VII* (2020).

[9] Plante, J., Gratadour, D., Bondonneau, L., and Viou, C., "A novel frb detection pipeline for nenufar," (2021).

Appendix B

Parset file

The following code snippet is an example of parset file used to describe experiments in NenuFAR's planning.

```
Observation.title=
Observation.name="SGR1935+2154"
Observation.contactName="VDECOENE"
Observation.contactEmail="valentin.decoene@subatech.in2p3.fr,nenufar-survey@obs-nancay.fr"
Observation.nrAnaBeams=1
Observation.nrBeams=3
Observation.nrPhaseCenter=0
Observation.topic=LT05 FAST_RADIO_BURSTS
Observation.calibration=calib.cur
Observation.cableDelays=ON
Observation.corAzel=enable
Observation.startTime=2023-09-08T16:00:00Z
Observation.stopTime=2023-09-08T20:00:00Z
Observation.parsetVersion=1.0


Output.sst_userfile=false
Output.bst_userfile=true
Output.xst_userfile=false
Output.hd_bitMode=8
Output.hd_lane0=192
Output.hd_lane1=192
Output.hd_lane2=192
Output.hd_nblocklane0=8
Output.hd_nblocklane1=8
Output.hd_nblocklane2=8
Output.hd_receivers=[undysputed,seti,radiogaga,codalema,thetis]

AnaBeam[0].target="SGR1935+2154"
AnaBeam[0].angle1=293.73163
AnaBeam[0].angle2=21.896700
AnaBeam[0].directionType=J2000
AnaBeam[0].startTime=2023-09-08T16:01:10Z
AnaBeam[0].duration=14320
AnaBeam[0].maList=[0,1,3,4,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,22,23,24,25,26,27,28,29,30,31,
AnaBeam[0].attList=[59,60,62,57,62,57,59,59,57,55,56,49,49,53,48,46,44,45,43,55,50,49,47,47,42,41,
AnaBeam[0].antList=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
AnaBeam[0].antState=ON
AnaBeam[0].filter=[3]
```

```
AnaBeam[0].filterTime=[2023-09-08T16:01:10Z]
AnaBeam[0].beamSquint=enable
AnaBeam[0].optFrq=60

Beam[0].target="SGR1935+2154_TF_LF"
Beam[0].toDo=DYNAMICSPECTRUM
Beam[0].parameters="TF: DF=1.52 DT=21.0 HAMM"
Beam[0].noBeam=0
Beam[0].angle1=293.73163
Beam[0].angle2=21.896700
Beam[0].directionType=J2000
Beam[0].startTime=2023-09-08T16:01:10Z
Beam[0].duration=14320
Beam[0].subbandList=[61..252]
Beam[0].lane0=[61..252]
Beam[0].lane0_ind=0

Beam[1].target="SGR1935+2154_TF_HF"
Beam[1].toDo=DYNAMICSPECTRUM
Beam[1].parameters="TF: DF=1.52 DT=21.0 HAMM"
Beam[1].noBeam=0
Beam[1].angle1=293.73163
Beam[1].angle2=21.896700
Beam[1].directionType=J2000
Beam[1].startTime=2023-09-08T16:01:10Z
Beam[1].duration=14320
Beam[1].subbandList=[253..444]
Beam[1].lane1=[253..444]
Beam[1].lane1_ind=0

Beam[2].target="SGR1935+2154_PULSAR"
Beam[2].toDo=PULSAR
Beam[2].parameters="SINGLE: --SEARCH --DSTIME=4096 --DEFARADAY --SRC=SGR1935+2154"
Beam[2].noBeam=0
Beam[2].angle1=293.73163
Beam[2].angle2=21.896700
Beam[2].directionType=J2000
Beam[2].startTime=2023-09-08T16:01:10Z
Beam[2].duration=14320
Beam[2].subbandList=[230..421]
Beam[2].lane2=[230..421]
Beam[2].lane2_ind=0
```

In the context of this work, we mainly used the `Beam[i].subbandList`, `Beam[i].startTime` and `Beam[i].duration` fields of this file. The start time and duration fields are self-explanatory, and `subbandList` is the list of frequency components requested. The format of the `subbandList` corresponds to the index of the components of a 1024-point FFT over a signal at 200 MHz. In more concrete terms, the actual frequency in Hz can be found by multiplying the values of `subbandList` by $\frac{200\,\text{MHz}}{1024}$.

Appendix C

Details on GPU architecture

In this chapter, we give details about some features of GPUs, and especially Nvidia GPUs, as well as some CUDA features.

C.1 GPU memory subunits

C.1.1 global memory

Global memory is the main memory system of the GPU. Physically, this memory is located on the side of the GPU, separated from the actual compute units, the SMs. It is generally a type of DRAM, such as Graphics Double Data Rate (GDDR)4/5/6 or High Bandwidth Memory (HBM).

Global memory is accessible for reading and writing for any thread of any kernel in the current context.

Accesses to global memory are generally required at some point, but are considered as slow compared to other types of memory subunits. Eliminating redundant global memory accesses is strongly encouraged, and recomputing a value is generally preferred to re-reading one.

Accesses to this memory subunit are faster when *coalesced*, meaning that successive threads access successive values in memory.

Wider memory accesses (8 or 16 B) also help approaching the maximum bandwidth. Reading multiple values as a wide memory access is commonly referred as a vectorized memory access.

C.1.2 local memory

Physically, local memory is located on the same piece of hardware than global memory. The only difference is that local memory is local to a thread, and cannot be accessed by other threads, even among the same CUDA kernel or CUDA block.

By default, it is cached through L2 and L1 cache, but passthroughs can be configured.

Accesses to local memory are generally considered as harmful, and removing them is often among the first optimization strategies. Indeed, local memory accesses generally arise when a lot of memory is required by each thread at once. If the amount of storage requested is greater than the capacity of an SM's register storage, registers spill in local memory.

Solutions typically rely on reducing the register pressure by better distributing the work across multiple threads of a CUDA block.

C.1.3 shared memory

Shared memory is a type of on-chip memory, located directly on each SM. It is a kind of programmable cache, shared between the threads of a CUDA block.

Accesses to shared memory can be considered as instantaneous, at least in the stage of optimization. They do not have to be coalesced, but care must be taken with bank conflicts.

C.1.4 constant memory

Constant memory is physically similar to global memory, with the exception that it is cached through a dedicated cache, the constant cache. It is read-only from CUDA kernels, and must be set using a special API from the host.

C.1.5 texture memory

Texture memory, is in turn similar to constant memory, being read-only from the device, and benefitting from a dedicated cache, the texture cache.

Moreover, it is not optimized for coalesced memory accesses, but for 2D-locality instead, enabling strongly different data distribution in the kernel. It also inherits from a number of historical GPU features regarding textures: a dedicated pointer arithmetic hardware unit, int to float conversion, interpolation and edge conditions.

C.1.6 surface memory

Surface memory if the last type of device memory. It is cached through texture cache, and benefits from the dedicated addressing hardware. It can be both read and written, but does not benefit from the other advantages of texture memory.

C.1.7 Other

We presented the six memory subunits of Nvidia GPUs, but this would not be complete without highlighting other components related to the topic of memory, even if they are not considered as subunits.

C.1.7.1 Register

Registers are required to perform computations: operands of an instruction must be in registers for the instruction to execute, and the result is stored in a register too.

Simply put, declaring a value in a kernel will result in a register being used to hold this value. In practice, the compiler tries to reuse registers as much possible, and there is no direct mapping between `C++` declarations and register usage. As long as the behavior of the program stays the same, the compiler is allowed to do virtually anything in its quest to the best performance.

On Nvidia GPUs, registers are one of the SM's resources, and different CUDA blocks compete for registers. Depending on the number of registers required by each CUDA block, multiple CUDA blocks may be able to run concurrently on a single SM. This is reflected by the *occupancy* metric of the Nsight Compute profiler.

C.1.7.2 host memory

Host memory is not located on the GPU, but can be made accessible to a CUDA kernel. When doing so, each memory transaction goes through the PCIe bus. This can be useful to avoid a bounce buffer, but must be used carefully as this is generally even slower than global memory.

C.1.7.3 peer memory

Similarly, memory from other GPUs can be mapped and accessed seamlessly. This suffers the same limitations as host memory accesses if the two GPUs are connected through the PCIe bus. However, much better performance can be achieved using the Nvlink proprietary interconnect, which can transfer up to 200 Gbit/s in each direction and per link, with support for multiple links depending on the GPU (V100: 6 links, A100: 12 links, H100: 18 links).

C.2 Compute Capability

Compute Capabilitys (CCs) describe the feature set of CUDA-enabled GPUs. Each CC exhibits a specific balance of maximum shared memory per block, number of SMs, and so on.

Additionally, some features from CUDA are only available for specific sets CC. Software fallback may be available, as is the case for the memory transfer API `memcpy_async`, hardware-accelerated from CC 8.0, but valid on other GPUs as well through software fallback.

Some others, such as Tensor Cores or TMA do not benefit from software fallback. The responsibility of using valid operations on a specific CC is left to the user, and can increase strongly code complexity.

C.3 Coalesced memory accesses

In CUDA-enabled GPUs, DRAM banks are 128 B wide. In order to read data from global memory, a full bank must filled.

When threads of a same block access small values (1–16 B) far from one another, these banks are filled completely for each memory request, and excessive data is not used. However, if threads access data with good locality, data in banks is reused and much less bank fills are needed to fullfil the memory transactions. This is called a *coalescing memory access*, and results in much better performance.

RÉSUMÉ

Les systèmes de notre monde sont en croissance constante en termes de précision, rapidité et consommation énergetique. Cette croissance passe par la production de volumes de données toujours plus importants, qui deviennent difficiles à traiter avec les technologies standard actuelles.
Dans le cadre de cette thèse, nous nous sommes intéressé à des technologies émergentes, telles que le calcul sur GPU, le réseau en espace utilisateur, DPDK et GPUDirect, permettant de répondre à ce besoin, et les avons appliquées à plusieurs projets concrets à fort impact scientifique et industriel (acquisition de front d'ondes pour optique adaptative, detection en temps réel de transients radio,traitement du signal radar) avec succès.

MOTS CLÉS

Instrumentation, GPU, Réseau, Traitement du signal, Optique adaptative, Radioastronomie, Radar

ABSTRACT

Systems of our world are following a constant growth in terms of precision, speed and electrical consumption. This growth involves the production of always bigger volumes of data, which become difficult to process using today's standard technologies.
In the context of this thesis, we looked into emerging technologies such as GPU computing, userland networking, DPDK and GPUDirect, which are able to answer this need, and applied them to multiple real-life projects with strong scientific and industrial impact (wavefront acquisition for adaptive optics, real-time fast radio burst detection, radar signal processing) with success.

KEYWORDS

Instrumentation, GPU, Networking, Signal processing, Adaptive optics, Radioastronomy, Radar