UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

MIMOPack: A High Performance Computing Library for MIMO Communication Systems

Doctoral Thesis

by

Carla Ramiro Sánchez

Supervisors:

Prof. Antonio M. Vidal Maciá
Prof. Alberto González Salvador

Valencia, Spain
June 2015

To my parents
Manuel and Marta

Abstract

Nowadays, several communication standards are emerging and evolving, searching higher transmission rates, reliability and coverage. This expansion is primarily driven by the continued increase in consumption of mobile multimedia services due to the emergence of new handheld devices such as smartphones and tablets.

One of the most significant techniques employed to meet these demands is the use of multiple transmit and receive antennas, known as MIMO (Multiple Input Multiple Output) systems. The use of this technology allows to increase the transmission rate and the quality of the transmission through the use of multiple antennas at the transmitter and receiver sides.

MIMO technologies have become an essential key in several wireless and broadband standards such as Wireless Local Area Network (WLAN), Worldwide interoperability for Microwave Acces (WiMAX), Long Term Evolution (LTE) and Next Generation Handheld (DVB-NGH), for the reception of Digital Terrestrial Television (DTT) in handheld devices. These technologies will be incorporated also in future standards, therefore is expected in the coming years a great deal of research in this field.

Clearly, the study of MIMO systems is critical in the current investigation, however the problems that arise from this technology are very complex. High Performance Computing (HPC) systems, and specifically, modern hardware architectures as multi-core and many-cores (e.g Graphics Processing Units (GPU)) are playing a key role in the development of efficient and low-complexity algorithms for MIMO transmissions. Proof of this is that the number of scientific contributions and research projects related to its use has increased in the last years. Also, some high performance libraries have been implemented as tools for researchers or companies involved in the development of future communication standards. Two of the most popular libraries are: IT++ that is a library based on the use of some optimized libraries for multi-core processors and the Communications System Toolbox designed for use with MATLAB and Simulink, which uses GPU computing. However, there is not a library able to run on a heterogeneous platform using all the available resources.

In view of the high computational requirements in MIMO application research and the shortage of tools able to satisfy them, we have made a

special effort to develop a library to ease the development of adaptable parallel applications in accordance with the different architectures of the executing platform. The library, called MIMOPack, aims to implement efficiently using parallel computing, a set of functions to perform some of the critical stages of MIMO communication systems simulation.

The main contribution of the thesis is the implementation of efficient Hard and Soft output detectors, since the detection stage is considered the most complex part of the communication process. These detectors are highly configurable and many of them include preprocessing techniques that reduce the computational cost and increase the performance.

The proposed library shows three important features: portability, efficiency and easy of use. This library can be run over the last generation of machine architectures (current realease allows GPUs and multi-core computation), or even simultaneously, since it is designed to use on heterogeneous machines exploiting the whole computational capacity thus reducing the response time of the most complex problems. The interface of the functions are common to all environments in order to simplify the use of the library, regardless of the machine where the functions will be executed. Moreover, some of the functions are callable from MATLAB increasing the portability of developed codes between different computing environments.

According to the library design and the performance assessment, we consider that MIMOPack may facilitate industrial and academic researchers the implementation of scientific codes without having to know different programming languages and machine architectures. This will allow to include more complex algorithms in their simulations and obtain their results faster. This is particularly important in the industry, since the manufacturers work to analyze and to propose their own technologies with the aim that it will be approved as a standard. Thus allowing to enforce their intellectual property rights over their competitors, who should obtain the corresponding licenses to include these technologies into their products.

Keywords: HPC Library, GPU, multi-core, CUDA, MIMO, Sphere decoding, Tree-Search detection.

Resumen

En la actualidad varios estándares de comunicación están surgiendo y evolucionando buscando velocidades de transmisión más altas, mayor fiabilidad y mejor cobertura. Esta expansión está principalmente impulsada por el continuo aumento en el consumo de servicios multimedia móviles debido a la aparición de nuevos dispositivos portátiles como los smartphones y las tabletas.

Una de las técnicas empleadas más importantes para satisfacer estas demandas es el uso de múltiples antenas de transmisión y recepción, conocida como sistemas MIMO (Multiple Input Multiple Output). El uso de esta tecnología permite aumentar la velocidad y la calidad de la transmisión a través del uso de múltiples antenas en el transmisor y en el receptor.

Las tecnologías MIMO se han convertido en una parte esencial en diferentes estándares inalámbricos y de banda ancha, tales como Wireless Local Area Network (WLAN), Worldwide Interoperability for Microwave Acces (WiMAX), Long Term Evolution (LTE) y Next Generation Handheld (DVB-NGH), para la recepción de la televisión digital terrestre (TDT) en dispositivos portátiles. Estas tecnologías se incorporarán también en futuros estándares, por lo tanto, se espera en los próximos años una gran cantidad de investigación en este campo.

Está claro que el estudio de los sistemas MIMO es crítico en la investigación actual, sin embargo los problemas que surgen de esta tecnología son muy complejos. La sistemas de computación de alto rendimiento, y en concreto, las arquitecturas hardware actuales como multi-core y many-core (p. ej. unidades de procesamiento gráfico (GPU)) están jugando un papel clave en el desarrollo de algoritmos eficientes y de baja complejidad en las transmisiones MIMO. Prueba de ello es que el número de contribuciones científicas y proyectos de investigación relacionados con su uso se han incrementado en el últimos años.

Además, algunas librerías de alto rendimiento se están utilizando como herramientas por investigadores o empresas involucradas en el desarrollo de futuros estándares de comunicación. Dos de las librerías más destacadas son: IT++ que es una librería basada en el uso de distintas librerías ya optimizadas para procesadores multi-core y el paquete Communications System Toolbox diseñada para su uso con MATLAB y Simulink, que utiliza com-

putación con GPU. Sin embargo, no hay una biblioteca capaz de ejecutarse en una plataforma heterogénea utilizando todos los recursos disponibles.

En vista de los altos requisitos computacionales en la investigación MIMO y la escasez de herramientas capaces de satisfacerlos, hemos implementado una librería que facilita el desarrollo de aplicaciones paralelas adaptables de acuerdo con las diferentes arquitecturas de la plataforma de ejecución. La librería, llamada MIMOPack, implementa de manera eficiente utilizando la computación paralela, un conjunto de funciones para llevar a cabo algunas de las etapas críticas en la simulación de un sistema de comunicación MIMO.

La principal aportación de la tesis es la implementación de detectores eficientes de salida Hard y Soft, ya que la etapa de detección es considerada la parte más compleja en el proceso de comunicación. Estos detectores son altamente configurables y muchos de ellos incluyen técnicas de preprocesamiento que reducen el coste computacional y aumentan el rendimiento. La librería propuesta tiene tres características importantes: la portabilidad, la eficiencia y facilidad de uso. Esta librería se puede ejecutar en la última generación de arquitecturas máquina. La versión actual permite computación en GPU y multi-core, incluso simultáneamente, ya que está diseñada para ser utilizada sobre plataformas heterogéneas que explotan toda la capacidad computacional reduciendo así el tiempo de respuesta de los problemas más complejos. Con el fin de simplificar el uso de la biblioteca, las interfaces de las funciones son comunes para todas las arquitecturas independientemente de la máquina donde serán ejecutadas. Por otra parte, algunas de las funciones se pueden llamar desde MATLAB aumentando la portabilidad de códigos desarrollados entre los diferentes entornos computacionales.

De acuerdo con el diseño de la biblioteca y la evaluación del rendimiento, consideramos que MIMOPack puede facilitar la implementación de códigos científicos a investigadores industriales y académicos sin tener que saber programar con diferentes lenguajes y arquitecturas. MIMOPack permitirá incluir algoritmos más complejos en las simulaciones y obtener los resultados más rápidamente. Esto es particularmente importante en la industria, ya que los fabricantes trabajan para analizar y proponer sus propias tecnologías lo antes posible con el objetivo de que sean aprobadas como un estándar. De este modo, los fabricantes pueden hacer valer sus derechos de propiedad intelectual frente a sus competidores, quienes luego deben obtener las correspondientes licencias si quieren incluir dichas tecnologías

en sus productos.

Palabras Clave: Librería HPC, GPU, multi-núcleo, CUDA, MIMO, Decodificación Esférica, detección por búsqueda en árbol.

Resum

En l'actualitat diversos estàndards de comunicació estan sorgint i evolucionant cercant velocitats de transmissió més altes, major fiabilitat i millor cobertura. Aquesta expansió, està principalment impulsada pel continu augment en el consum de serveis mòbils multimèdia a causa de l'aparició de nous dispositius portàtils com els smartphones i les tablets. Una de les tècniques més importants utilitzades per a satisfer aquestes demandes és l'ús de múltiples antenes de transmissió i recepció, coneguda com sistemes MIMO (Multiple Input Multiple Output). L'ús d'aquesta tecnología permet augmentar la velocitat de transmissió i la qualitat de transmissió a través de l'ús de múltiples antenes en el transmissor i en el receptor.

Les tecnologies MIMO s'han convertit en una part essencial en diferents estàndards inalàmbrics i de banda ampla, tals com Wireless Local Area Network (WLAN), Worldwide Interoperability for Microwave Acces (WiMAX), Long Term Evolution (LTE) i Next Generation Handheld (DVB-NGH), per a la recepció de Televisió Digital Terrestre (TDT) en dispositius portàtils. Aquestes tecnologies s'incorporaran també en futurs estàndards, per tant, s'espera en els pròxims anys una gran quantitat d'investigació en aquest camp.

Està clar que l'estudi dels sistemes MIMO és crític en la recerca actual, no obstant açó, els problemes que sorgeixen d'aquesta tecnologia són molt complexos. Els sistemes de computació d'alt rendiment, i en concret, les arquitectures hardware actuals com multi-core i many-core (p. ej. unitats de processament gràfic (GPU)), estan jugant un paper clau en el desenvolupament d'algoritmes eficients i de baixa complexitat en les transmissions MIMO. Prova d'açò és que el nombre de contribucions científiques i projectes d'investigació relacionats amb el seu ús s'han incrementat en els últims anys.

A més, algunes llibreries d'alt rendiment estan utilitzant-se com a eines per investigadors o empreses involucrades en el desenvolupament de futurs estàndards de comunicació. Dos de les llibreries més destacades són: IT++ que és una llibreria basada en lús de diferents llibreries ja optimitzades per a processadors multi-core i el paquet Communications System Toolbox dissenyat per al seu ús amb MATLAB i Simulink, que utilitza computació amb GPU. No obstant açò, no hi ha una biblioteca capaç d'executar-se en

una plataforma heterogènia utilitzant tots el recursos disponibles.

Degut als alts requisits computacionals en la investigació MIMO i l'escacès d'eines capaces de satisfer-los, hem implementat una llibreria que facilita el desenvolupament d'aplicacions paral·leles adaptables d'acord amb les diferentes arquitectures de la plataforma d'ejecució. La llibreria, anomenada MIMOPack, implementa de manera eficient utilitzant la computació paral·lela, un conjunt de funcions per dur a terme algunes de les etapes crítiques en la simulació d'un sistema de comunicació MIMO.

La principal aportació de la tesi és la implementació de detectors eficients d'exida Hard i Soft, ja que l'etapa de detecció és considerada la part més complexa en el procés de comunicació. Estos detectors són altament configurables i molts d'ells inclouen tècniques de preprocessament que redueixen el cost computacional i augmenten el rendiment. La llibreria proposta té tres característiques importants: la portabilitat, l'eficiència i la facilitat d'ús. Aquesta llibreria pot executar-se en l'última generació d'arquitectures màquina. La versió actual permet computació en GPU i multi-core, fins i tot simultàniament, ja que està dissenyada per a ser utilitzada sobre plataformes heterogènies que exploten tota la capacitat computacional reduint així el temps de resposta dels problemes més complexos. Amb el fi de simplificar l'ús de la biblioteca, les interfaces de les funcions són comunes per a totes les arquitectures independentment de la màquina on seran executades. D'altra banda, algunes de les funcions poden ser utilitzades des de MATLAB augmentant la portabilitat de còdics desenvolupats entre els diferentes entorns computacionals.

D'acord amb el disseny de la biblioteca i l'evaluació del rendiment, considerem que MIMOPack pot facilitar la implementació de còdics científics a investigadors industrials i acadèmics sense haver de saber programar amb diferents llenguatges i arquitectures. MIMOPack permetrà incloure algoritmes més complexos en les seues simulacions i obtindre els seus resultats més ràpid. Açò és particularment important en la industria, ja que els fabricants treballen per a analitzar i proposar les seues pròpies tecnologies el més prompte possible amb l'objectiu que siguen aprovades com un estàndard. D'aquesta menera, els fabricants podran fer valdre els seus drets de propietat intel·lectual enfront dels seus competidors, els qui després han d'obtenir les corresponents llicències si volen incloure aquestes tecnologies en els seus productes.

Paraules Clau: Llibreria HPC, GPU, multi-nucli, CUDA, MIMO, Sphere Decoding, detecciò per recerca en arbre.

Acknowledgements

It is a pleasure for me to thank those who made this thesis possible. First and foremost, I offer my sincerest gratitude to my supervisors, Prof. Antonio Vidal and Prof. Alberto González, who supported me throughout this thesis with their knowledge and advice whilst allowing me the room to work in my own way.

I am very grateful to Prof. Baltasar Beferull from the University of Agder, Dr. José Miguel Mantas from the University of Granada and Dr. Leroy Anthony Drummond from Lawrence Berkeley National Laboratory for serving as reviewers of this thesis and for providing me with very useful comments that helped to improve the final manuscript.

I would like to thank Dr. Alain Mourad, who hosted me at the Samsung Electronics Research Institute, Staines, United Kingdom. I really appreciate the opportunity he gave me and his kind support during the months I spent working with his team.

I would like to show my gratitude to all the people at the Universitat Politècnica de València that shared my daily work at the Department of Information Systems and Computation (DSIC) and at the Institute of Telecommunications and Multimedia Applications (iTEAM). In particular, I would like to thank Dr. Pedro Alonso, Dr. Victor García and Dr. Francisco Martínez for their support and collaboration. Thanks also to my current and former colleagues at the iTEAM: Jose A. Belloch, Murilo Boratto, Fernando Domene, Sandra Roger and Marian Simarro.

Thanks to all my friends for all the support and for making me smile even in my worse days.

I would like to acknowledge the help and encouragement given by my parents, Manuel and Marta, for making me be who I am, for their endless love, sacrifices and advices. Thanks to my dear sisters Marta and Patricia and also to the rest of my family, especially to my brother in law Ramón and my nephew Álvaro for their fondness and for supporting me all the way.

Last, but not least, I would like to express my sincere gratitude to Piedad, for her invaluable emotionally support and for always being by my side. You complete me in every way.

Carla Ramiro Sánchez
June 2015

Contents

List of Figures

List of symbols

$\mathbf{X}$	Matrix
$\mathbf{x}$	Vector
x	Scalar
$X_{i,j}$	i,j component of matrix $\mathbf{X}$
$X_{i,:}$	i-th row of matrix $\mathbf{X}$
$X_{:,i}$	i-th column of matrix $\mathbf{X}$
$X_{i:j,k:l}$	Elements from i-th to j-th row and from k-th to l-th column of $\mathbf{X}$
x_i	i-th component of vector $\mathbf{x}$
$(\cdot)^{\mathrm{T}}$	Transpose
$(\cdot)^*$	Complex conjugation
$(\cdot)^{\mathrm{H}}$	Conjugate transpose
$(\cdot)^\dagger$	Moore-Penrose pseudoinverse
$(\cdot)^{-1}$	Matrix inversion
$\mathbf{I}_{N,N}$	Identity matrix of size $N \times N$
$\lvert \cdot \rvert$	Absolute value
$\lVert \cdot \rVert_p$	ℓ_p norm
$\lVert \cdot \rVert$	2 norm
$P(A)$	Marginal probability of A
$P(A\lvert B)$	Conditional probability of A, given B
$\Re\{\cdot\}$	Real part of a complex number
$\Im\{\cdot\}$	Imaginary part of a complex number
max	Maximum of a set
min	Minimum of a set
arg max	Argument of the maximum of a set
arg min	Argument of the minimum of a set
log	Natural logarithm
$\mathcal{Q}\{\cdot\}$	Quantization (slicing) operation
$\Theta\{\cdot\}$	Denotes an asymptotic complexity order of $\cdot$
$[\cdot]$	Denotes the binary-valued representation of the constellation symbol $\cdot$

Abbreviations and Acronyms

ASD	Automatic Sphere Decoder
ASIC	Application Specific Integrated Circuits
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BICM	Bit-Interleaved Coded-Modulation
BLAST	Bell Labs Layered Space-Time
BPSK	Binary Phase Shift Keying
BS	Base Station
CUDA	Compute Unified Device Architecture
CVP	Closest Vector Problem
DF	Decision Feedback
DSP	Digital Signal Processor
DTT	Digital Terrestrial Television
ED	Euclidean Distance
e.g.	for example (from the latin exempli gratia)
FE	Full Expansion
FPGA	Field Programmable Gate Array
FP-SD	Fincke-Pohst Sphere Decoder
FPFSD	Fully-Parallel Fixed-complexity Sphere Decoder
FSD	Fixed-complexity Sphere Decoder
GPU	Graphics Processing Unit
HPC	High Performance Computing
LDPC	Low-Density Parity-Check
i.e.	that is (from the latin id est)
i.i.d.	Independently, identically distributed
LLR	Log-likelihood ratio
LTE	Long Term Evolution
MAP	Maximum-A-Posteriori
MIMO	Multiple-Input Multiple-Output
MISO	Multiple-Input Single-Output
ML	Maximum-Likelihood
MLE	Maximum-Likelihood Exhaustive
MMSE	Minimum Mean Squared Error

MSs	Single-antenna users
MU	Multiuser
NGH	Next Generation Handheld
OFDM	Orthogonal Frequency Division Multiplexing
OSIC	Ordered Successive Interference Cancellation
PED	Partial Euclidean Distance
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
SD	Sphere Decoder
SDR	Software Defined Radio
SE	Single-path Expansion
SESD	Schnorr-Euchner Sphere Decoder
SFSD	Soft-output Fixed-complexity Sphere Decoder
SIC	Successive Interference Cancellation
SISO	Single-Input Single-Output
SIMO	Single-Input Multiple-Output
SM	Stream Multiprocessor
SNR	Signal to Noise Ratio
THP	Tomlinson-Harashima Precoding
WiMAX	Worldwide Interoperability for Microwave Access
WLAN	Wireless Local Area Network
ZF	Zero Forcing
ZFSIC	Zero Forcing with Successive Interference Cancellation

1 Introduction and Objectives

Introduction and Objectives 1

1.1 Background

Multiple-Input multiple-Output (MIMO) systems have a huge impact in modern wireless communications, since they allow to increase the reliability, coverage and tranmission rates without the need for extra bandwidth or power cost [1][2]. To boost the data rates of current generation cellular networks, MIMO technologies have been adopted by many wireless standards such as Long Term Evolution (LTE), Worldwide interoperability for Microwave Access (WiMAX), Wireles Local Area Network (WLAN) and also by broadband standards such as Digital Video Broadcasting Next Generation Handheld (DVB-NGH). The influence of the new handheld devices such as smart phones and tablets is also important, since it has caused a drastic growth of mobile multimedia services. Hence, MIMO surely will become an imperative technology of wireless communication systems to increase the data traffic capacity.

MIMO systems emerged as the evolution of the classic and simplest Single-Input Single-Output (SISO) systems (see Fig. 1.1), which use one transmitting antenna and one receiving antenna. The first advance towards

MIMO was to employ more than one transmitting antenna at the base station and one receiving antenna such as the Multiple-Input Single-Output (MISO), or one transmitting antenna and several antennnas at the receiver side such as Single-Input Multiple-Output (SIMO). Finally, the complete growth is given by using several transmitting and receiving antennas in the same system (MIMO). This configuration increases the link reliability due to antenna diversity gain and the spectral efficiency through spatial-multiplexing gain.

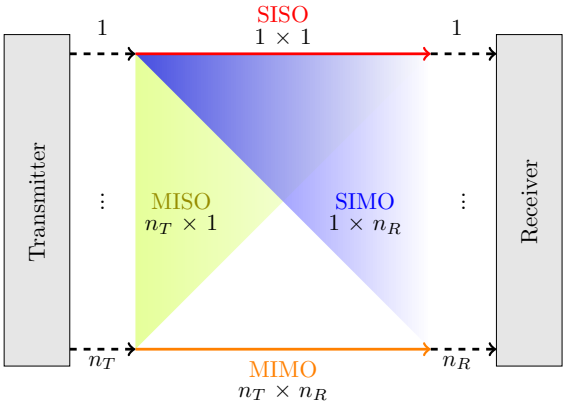


Figure 1.1. The different diversities of the MIMO systems evolution.

MIMO technologies are also used to improve the performance of Orthogonal Frequency Division Multiplexing (OFDM) systems. OFDM has become a popular method which converts a frequency-selective channel into a set of frequency flat subchannels. The combination of OFDM with MIMO systems, known as MIMO-OFDM [3], uses very efficiently the available bandwidth, since it allows transmitting different streams over different subcarriers and different spatial beams in each one of the subcarriers by using MIMO precoding. The scalability in the number of subcarriers per MIMO-OFDM symbol is a key factor in wireless standards [4].

However, the use of MIMO technologies involves an increment of the detection process complexity. The detector is present at the receiver side and is the responsible for recovering the received signals (which are affected by the channel fluctuation) with the maximum reliability. This step becomes

in many cases the most complex stage in the communication. The number of transmit and receive antennas is another important factor that affects the performance of a MIMO system, because as the system grows the communication process becomes more complicated. Although the number of antennas currently allowed in the standards is not large, it is expected that in the near future more than 100 transmit antennas could be used [5][6]. All these reasons motivate the search for high-throughput versatile receiver implementations suitable to be reconfigured with the system parameters.

Practical implementation of MIMO receiver schemes and *Software Defined Radio* (SDR) platforms have been traditionally developed using *Digital Signal Processors* (DSP) [7], *Field Programmable Gate Arrays* (FPGA) or *Application Specific Integrated Circuits* (ASIC) [8][9]. The last advances in computational hardware (multi-core and Graphic Processing Units) have allowed the development of high-throughput implementations. In last years, the number of scientific contributions and research projects related to the use of High Performance Computing (HPC) systems has significantly increased. This phenomenon has occurred in almost all engineering fields that require intensive computing, and Signal Processing is not an exception [10].

The use of the last generation of HPC systems such as multi-core CPUs and Graphics Processing Units (GPUs) has become attractive for the efficient implementation of parallel signal processing algorithms with high computational requirements, such as the scheme reported in [11], high-throughput MIMO detectors [9][12] and fast LDPC decoders [13]. As is expressed in [14], "signal processing algorithm designers of the future will need to understand better the nuances of multi-core computing engines". In that special issue, several authors describe "novel applications that can be enabled by platforms with multiple cores, and more extensive design examples of signal processing on platforms with multiple cores that demonstrate useful techniques for developing efficient implementations". Moreover, nowadays, DSP architectures are incorporating multi-core capacities [15]. The implementation of advanced algorithms able to use both architectures is crucial in MIMO research, since it allows to fully exploit the capabilities of the modern machine architectures and to reduce the response time of computationally expensive problems.

Currently, the simultaneous use of different types of architectures (GPUs and multi-core) on a single system enjoys great popularity and is currently used, for example, by numerical linear algebra libraries as MAGMA [16] or

CULA [17]. Nevertheless, in the field of communication systems applications, few tools or high performance libraries are available. An exception is the Communications System Toolbox [18] of MATLAB that provides algorithms for designing, simulating, and analyzing communications systems. Although this software is excellent and widely used by the scientific community, just a small set of functions are prepared to use parallel computing with GPUs. Other library is IT++ [19], which is a C++ library of mathematical, signal processing and communication classes and functions. IT++ uses a set of some existent libraries (BLAS, LAPACK, FFTW) to increase its functionality and speed. However, this library is oriented to its exclusive use on shared-memory processors; it does not have support to use in GPUs.

In view of the high computational requirements in MIMO research and the shortage of tools able to satisfy them, in this thesis a special effort have been made to develop a library hoping to ease the development of adaptable parallel applications in accordance with the different architectures of the execution platform. The library, called MIMOPack, aims to implement efficiently, using parallel computing, a set of functions to perform some of the critical stages in MIMO communication systems. This library can be run over the last generation of machine architectures (e.g GPUs and multicore), or even simultaneously, since it is designed to use on heterogeneous machines to exploit the whole computational capacity thus reducing the response time of the most complex problems.

1.2 Motivation

The use of MIMO technology has had enormous repercussion in today's telecommunications systems and surely it will keep on doing it in the near future. The benefits offered are achieved at the expense of an increase in the material costs to deploy multiple antennas at both the transmitter and the receiver, and also at the expense of additional complexity at the receiver end of the MIMO system. For this reason, signal detection has been the subject of deep study during the last decade and the search for high throughput practical implementations remains essential today.

The goal is to develop fast algorithms to optimize the design and the validation process of new MIMO schemes and technologies. Then, this work aims to contribute to meet this goal. This section describes some particular

motivations leading towards the design of the high performance library.

Clearly, the use of HPC systems brings big benefits, but it will also sets big challenges. In recent years, a large variety of machine architectures have appeared; in view of this situation researchers of the scientific community are forced to write codes in different programming languages and consider many details of the architecture in order to use efficiently the whole target system. High performance computing libraries are essential to facilitate the implementation of scientific codes on a widespread range of architectures.

Furthermore, there are several important entities involved in the development of new communication standards: administrations, network operators, manufacturers, users, research bodies, universities, consultancy companies, partnerships and others [20]. The objective of a standard is to provide a set of rules, guidelines or characteristics ensuring the interoperability between systems developed by different manufacturers. These manufacturers work to propose their own technologies with the aim that it will be approved as a standard. This would allow to enforce their intellectual property rights over its competitors, who should then obtain the corresponding license to include the technology adopted as standard into their products. In many cases, simulation is the only way to get these proposals but these simulations often involve a large computational burden, since they try to simulate the transmission of large amount of bits in order to obtain results close to those that would be obtained in a real transmission. Normally these simulations require weeks or even months to be completed. Thereby, MIMOPack may allow the launch of large simulations, opening the door for industrial researchers to analyze their technologies faster than through conventional simulation, and hence obtain more patent opportunities than their potential competitors.

1.3 Objectives

Taking into account the above presented motivations, the main goals of this thesis are the following (see Figure 1.2):

- To develop an efficient library of functions able to perform some of the most important and complex stages in a MIMO communication system which are described below:

  - *Hard-Output Detection*: The detector becomes often the most computationally expensive algorithm within a MIMO receiver if nearly optimal data detection is desired. The detector is responsible of processing the received mixture of signals affected by the channel in order to recover the transmitted data with the accuracy required by the considered application. This issue motivates the search for high-throughput MIMO hard-output detectors capable to be reconfigured and scalable with the system parameters. Also, some strategies such as channel matrix preprocessing techniques can be used in order to eventually decrease the computational cost of data detection.
  - *Error control coding and Soft-Output Detection*: Error control coding ensures the desired quality of service for a given data rate and it is necessary to improve reliability of MIMO systems. Therefore, the search of a good combination of detection MIMO schemes and coding schemes have drawn attention in recent years. The most promising coding schemes are Bit-Interleaved Coded Modulation (BICM) [21]. The information bits are encoded at the transmitter using an error-correction code. The soft demodulator provides the reliability information in terms of real valued log-likehood ratios (LLR). These values are used by the channel decoder to make final decisions on the received coded bits. Nevertheless, these sophisticated techniques cause a significant increase in the computational cost and require large computational power.
  - *MIMO precoding*: In the downlink scenario of multi-user (MU) MIMO transmissions, a base station equipped with multiple antennas transmits information to several independent users. The detection process becomes more complex due to the absence of cooperation between the users. In order to simplify the detector complexity at the receiver side, several precoding techniques were devised by various authors.

- To contribute with high-throughput implementations of functions using parallel processing, to evaluate them in terms of execution time, speedup and scalability, to compare them with other existing implementations.

- To facilitate to the programmer the implementation of codes on a wide

range of architectures, incrementing the portability of codes between different computing platforms by using common interfaces for all the considered environments. This approach simplifies the use of the library, regardless of the machine where it will be executed.

- To develop a set of highly configurable functions able to be executed with different parameters:
  - Simulation setup: constellation size, number of transmitter and receiver antennas, channel conditions etc.
  - Execution platform setup: different kind and quantity of computational resources to be used during the execution.

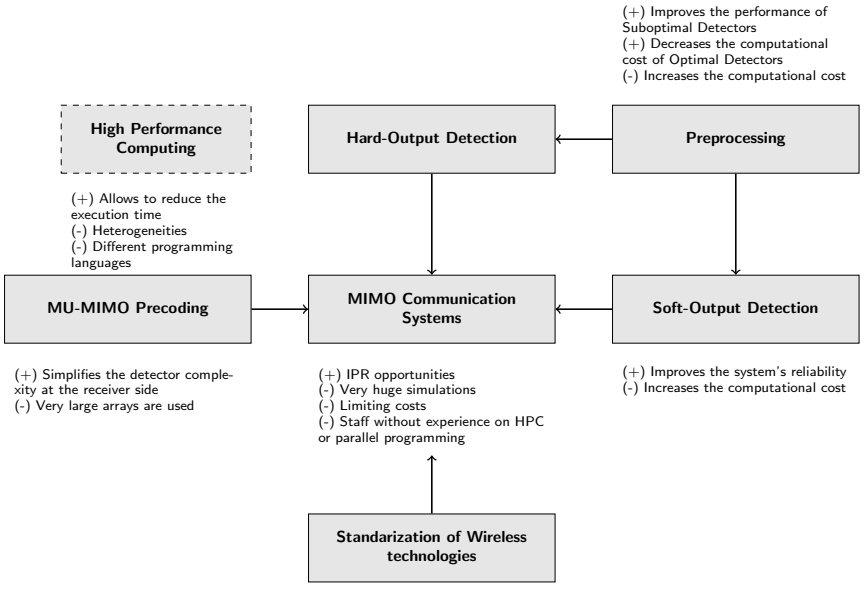


Figure 1.2. Nature and scope of the thesis.

1.4 Key Contributions

The main contribution of the thesis is the implementation of efficient Hard and Soft output detectors, since the detection stage is considered the most

complex part of the communication process. These detectors are highly configurable and many of them include preprocessing techniques that reduce the computational cost and increase the performance. The library has been designed to fulfill the following goals:

- Portable: This library can be run over the last generation of machine architectures (current release allows GPUs and multi-core computation), or even simultaneously, since it was designed to be used on heterogeneous machines exploiting the whole computational capacity. Moreover, some of functions are callable from MATLAB, increasing the portability of developed codes between different computing environments.
- Efficient: The library is composed of highly parallelized functions. Each function is parallelized taking into account the characteristics of the hardware device where it will be executed.
- User friendly: The interface of the functions is common to all environments in order to simplify the use of the library, regardless of the machine where it will be executed. This feature allows MIMOPack users to generate their codes without burdening the programmer with the drawbacks of multi-core or GPU development.

MIMOPack can be especially useful for industrial and academic researchers. The library eases the implementation of scientific codes without having to know different programming languages and the machine architecture in depth. This should allow to include more complex algorithms in their simulations and obtain their results faster. This is particularly important in the industry, since the manufacturers work to analyze and to propose their own technologies with the aim that it will be approved as a standard. Thus allowing to enforce their intellectual property rights over their competitors, who should obtain the corresponding licenses to include these technologies into their products.

1.5 Organization of the Thesis

This thesis is structured in seven sections that describe the developed research. It is important to remark that this thesis involves two different

disciplines: Signal Processing and Computational Science, consequently, it has been deemed appropriate to introduce basic concepts of both fields in order to ease the reading, by specialists of each field. The chapters are arranged and presented as follows:

- Chapter 2: The first part of the chapter contains some preliminaries related to the MIMO systems. The coded transmission in a MIMO-BICM is presented and, accordingly, the concept of soft demodulation is introduced. Furthermore, the signal precoding technique in multiuser MIMO systems are also reviewed. Next, different methods used in the main stages of the transmission chain are investigated and also how they have been addressed and computed up to now. Finally, the available High Performance Computing software packages related to the simulation of communication systems are evaluated.

- Chapter 3: This chapter presents a detailed description of some tools and optimizations used to speed up the algorithms and to reduce the computational complexity of the MIMO simulation. First, some hardware and software tools used for the development of the library are described. The next part of the chapter presents an overview of the heterogeneous computational model assumed for the library design. Finally, a method to decrease the complexity of the MIMO detectors consisting in an efficient calculation of the Euclidean distances is proposed.

- Chapter 4: In this chapter, the implementation of several Hard-Oputput detectors are presented. The chapter includes a detailed description of the sequential and parallelized algorithms. Some channel matrix orderings are proposed to reduce the complexity of the detectors. In addition, the execution times of all the proposed OpenMP and GPU parallel implementations are compared to the unoptimized implementation (i.e. without the EEDC optimization).

- Chapter 5: This chapter is focused on the implementation of Soft-Output detectors in MIMO-BICM. First, the optimum MAP detector and the Max-Log Approximation are presented and evaluated with different parameters. In the last part two efficient fixed-complexity demodulators are proposed and implemented. These detectors are based on a list extension of the Hard-Output FSD scheme. The

computational cost, some aspects and options of parallelization with OpenMP and CUDA are analized. Furthermore, numerical results are showed in order to assess the efficiency of the parallel implementations.

- Chapter 6: This chapter contains the library design and documentation. The main features of the MIMOPack library are explained. Additionally, a representative example of a practical MIMO communication system simulation with MIMOPack routines is presented.

- Chapter 7: This chapter contains the main conclusions and publications derived from the work developed in this thesis. Finally, some possible future research lines are given.

State-of-the-Art 2

State-of-the-Art 2

The simulation of MIMO transmission systems is essential to progress in the design and development of new technologies since it allows to analyze and test the new proposals without the need for its implementation in a physical system. Generally, it is necessary to simulate the transmission of many bits to obtain meaningful results, i.e. as nearly alike as possible to those obtained in a real service reception. This practice yields a large computational burden, therefore modeling and simulation using efficient techniques and high performance computing have made a significant contribution to advances in MIMO technologies.

The communication process in a MIMO system can be represented as a set of interconnected functional blocks. Not all stages have the same computational cost, indeed, some of them can become the 80% of the whole simulation cost. The state-of-the-art of the most complex stages is reviewed in the following sections and how they have been addressed and computed up to now. We can differentiate between two types of contributions primarily devised to reduce the complexity of the detection. On the one hand, theoretical contributions are those that attempt to reduce the complexity of the method from a theoretical point of view. For example: reducing the search range, applying channel matrix preprocessing techniques, etc. On the other hand, a practical contribution seeks to reduce the computational

cost of the method by its implementation for efficient processing thereof, e.g. multi-core or GPU implementations.

After some preliminaries related to the MIMO systems, the first part of the chapter enumerates different methods and techniques used in the main stages of the transmission chain. Finally, the available High Performance Computing software packages related to the development of communication systems are evaluated.

2.1 System Overview

The well-known *Bell-Labs layered Space-Time system* (BLAST) is a high speed wireless communication system that employs multiple antennas at both the transmitter and the receiver [22]. Figure 2.2 comprises a block diagram depiction of a MIMO-BLAST system equipped with n_T transmitter antennas and $n_R \geq n_T$ receive antennas. Spatial multiplexing gain is achieved by splitting the data bitstream into n_T transmit antennas. The data is simultaneously sent to the channel, thus overlapping in both time and frequency. The signals are received by n_R receiving antennas, as shown in Fig. 2.2, and the receiver has the task of processing the received signals in order to recover the transmitted data.

The sequence of information bits is spatially demultiplexed into the n_T antenna streams ("layers"). In each layer, groups of m bits are mapped to complex data symbols $s_i[n] \in \mathcal{O}$ with $i = 1, \ldots, n_T$. Here, $\mathcal{O}$ denotes the finite symbol alphabet or constellation. Figure 2.1 shows the quadrature-amplitud-modulation (QAM) constellations usually employed in MIMO communications, which are known as M-QAM. M stands for the number of constellation points (constellation size) and belongs usually to the set $M = \{2, 4, 16, 64\}$. We can define the constellation as $\mathcal{O} = \{a + bj : a, b \in PM\}$ where PM represents the real-valued representacion of the constellation. Then, symbols s_i are taken from $\mathcal{O}$ and carry $m = \log_2 M$ Gray-encoded bits each, which in a BLAST system are grouped after being demultiplexed into n_T streams. The transmitted vector at time instant n is denoted by $\mathbf{s}[n] = (s_1[n], s_2[n], \ldots, s_{n_T}[n])^T$ and carries $m \cdot n_T$ bits. The baseband equivalent model for the received vector $\mathbf{y}[n] = (y_1[n], y_2[n], \ldots, y_{n_R}[n])^T$ is given by

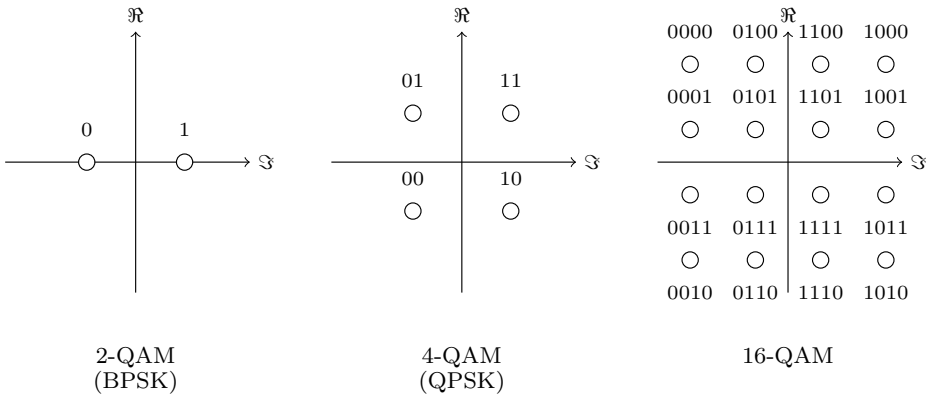


Figure 2.1. Most employed PSK and M-QAM constellations.

$$\mathbf{y}[n] = \mathbf{H}[n]\mathbf{s}[n] + \mathbf{v}[n], \quad n = 1, \ldots, N_c \tag{2.1}$$

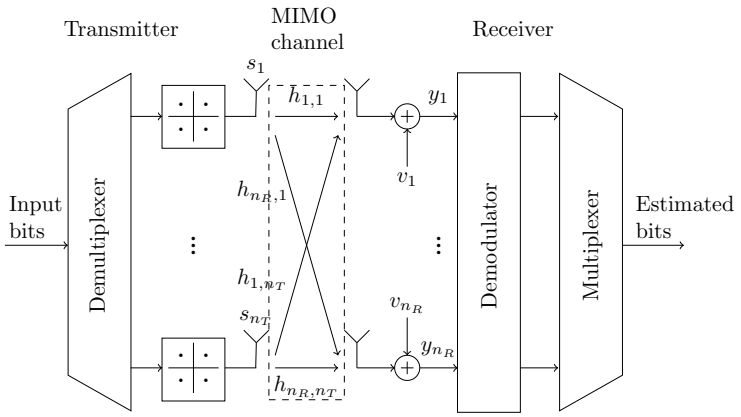


Figure 2.2. MIMO system model with n_T transmit antennas and n_R receive antennas with QPSK symbols.

where N_c is the number of time instants in the entire transmission. Here, $\mathbf{H}[n]$ is an $n_R \times n_T$ matrix modeling the Rayleigh fading MIMO channel which is assumed to be perfectly known at the receiver and remains constant over a symbol-time block of lenght L_{ch}. The channel matrix is formed by

$n_R \times n_T$ elements, $h_{i,j}$, which represents the fading gain between the j-th transmit antenna and the i-th receive antenna:

$$\mathbf{H} = \begin{bmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,n_T} \\ h_{2,1} & h_{2,2} & \dots & h_{2,n_T} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n_R,1} & h_{n_R,2} & \dots & h_{n_R,n_T} \end{bmatrix}. \tag{2.2}$$

The components of noise vector $\mathbf{v}[n] = (v_1[n], v_2[n], \dots, v_{n_R}[n])^T$ are assumed independent and circularly symmetric complex Gaussian with variance σ_w^2. For simplicity of notation, we will omit the symbol-time index n in order to ease the understanding of the rest of the chapter. Then, the equivalent model for received vector is given by

$$\mathbf{y} = \mathbf{H}\mathbf{s} + \mathbf{v}. \tag{2.3}$$

For practical reasons, the $(n_R \times n_T)$-dimensional complex equation (2.3) is often transformed into an equivalent $(2n_R \times 2n_T)$-dimensional real-valued representation as described in [23]:

$$\begin{bmatrix} \Re(\mathbf{y}) \\ \Im(\mathbf{y}) \end{bmatrix} = \begin{bmatrix} \Re(\mathbf{H}) & -\Im(\mathbf{H}) \\ \Im(\mathbf{H}) & \Re(\mathbf{H}) \end{bmatrix} \begin{bmatrix} \Re(\mathbf{s}) \\ \Im(\mathbf{s}) \end{bmatrix} + \begin{bmatrix} \Re(\mathbf{v}) \\ \Im(\mathbf{v}) \end{bmatrix} \tag{2.4}$$

where $\Re(\cdot)$ and $\Im(\cdot)$ denote the real and imaginary part of $(\cdot)$, respectively.

This is the simplest MIMO system, however nowadays more sophisticated transmission schemes are being used such as *Bit Interleaved Coded Modulation* (BICM) [21] and *Multiuser* (MU) MIMO [24] systems, which are introduced in the following sections. These advanced systems involve the use of new and more complex methods (e.g channel coding, soft-output detectors, precoding techniques) and have motivated the search for high-throughput versatile receiver implementations.

2.1.1 Encoded Transmissions

Error control coding is necessary to improve reliability of the MIMO system especially when we are using low-quality channels. Therefore, the search of a good combination of detection MIMO schemes and coding schemes have

drawn attention in recent years. The most promissing coding scheme is *Bit-Interleaved Coded Modulation* (BICM) [21][25].

In a MIMO-BICM system (such as the one shown in Fig. 2.3), a block of information bits $\mathbf{b}$ is encoded using an error correcting code before being demultiplexed into n_T layers. The coded bits are then passed through a bitwise interleaver Π generating a pseudo-random sequence $\mathbf{c}$ and mapped into symbols via Gray labeling.

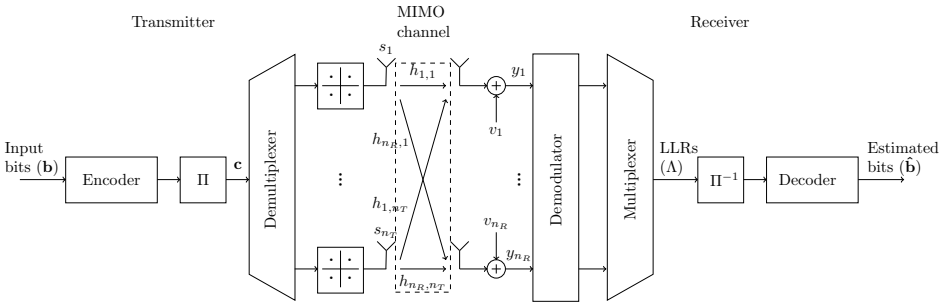


Figure 2.3. Block diagram of a MIMO-BICM system with QPSK symbols.

At the receiver side, the soft detector uses the model (2.3) to calculate the soft information about the code bits in terms of log-likelihood ratios *LLRs* (Λ). Thus, the detector uses the received vector $\mathbf{y}$ and the channel matrix $\mathbf{H}$, to calculate $\Lambda_{i,k}$ for each coded bit $c_{i,k}$, with $i = 1, \ldots, n_T$ and $k = 1, \ldots, m$, of the sent symbol vector $\mathbf{s}$. Finally, the reliability information (LLRs) is de-interleaved (Π^{-1}) and multiplexed into a single stream which will be used by the channel decoder. The soft demodulator computes the *LLRs* according to the following ratio of conditional bit probabilities

$$\Lambda_{i,k} = \log \frac{\mathrm{P}(c_{i,k} = 1|\mathbf{y}, \mathbf{H})}{\mathrm{P}(c_{i,k} = 0|\mathbf{y}, \mathbf{H})} \tag{2.5}$$

where $c_{i,k}$ denotes the k-th bit of symbol s_i and $\mathrm{P}(c_{i,k} = u|\mathbf{y}, \mathbf{H})$ denotes the probability that the coded bit $c_{i,k}$ takes the value u, taking into account $\mathbf{H}$ and $\mathbf{y}$. These LLRs are deinterleaved using Π^{-1} and used by the channel decoder to make final decisions about the transmitted sequence bits $\hat{\mathbf{b}}$.

2.1.2 Multiuser Scenario

In a downlink multiuser MIMO-OFDM system [24], the base station (BS) equipped with N antennas communicates to $K \leq N$ single-antenna users (MSs) over M subcarriers (such as the one shown in Fig. 2.4). The received signal $\mathbf{y}$ for the K users at the subcarrier m can be expressed as:

$$\mathbf{y}[m] = \mathbf{H}[m]\mathbf{s}[m] + \mathbf{v}[m] \tag{2.6}$$

where $\mathbf{y}[m] = [y_1[m], \ldots, y_K[m]]^T$ contains the received symbols for each MS and $\mathbf{v}[m] = [v_1[m], \ldots, v_K[m]]^T$ is the received noise for the m-th subcarrier.

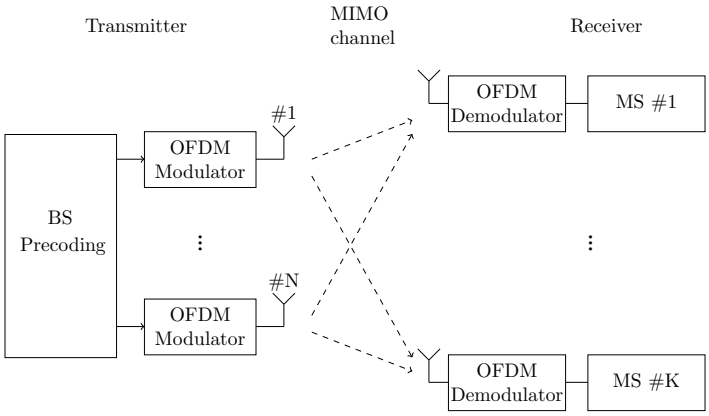


Figure 2.4. Block diagram of a MU-MIMO system with a Base Station (BS) with N antennas and K single-antenna users (MSs).

In the downlink scenario, the BS station sends information to each user, who receives the information of the K users. However the MS exclusively must process its own information; i.e. the MSs cannot cooperate together. This situation difficults the detection process. In order to reduce the complexity at the receiver side and then reduce manufacturing costs and energy consumption, several preprocessing techniques (precoding) emerged to move the computational complexity to the BS, which simplifies the mobile stations.

2.2 System Architecture

A generic MIMO system block diagram is shown in Figure 2.5. This diagram comprises some of the most important modules used in the transmission over the systems seen in the previous section. The most complex stages are depicted in blue and the blocks required to simulate a BICM MIMO system are represented with dashed lines. Therefore, not all modules must be used for the simulation of MIMO transmission, but rather depending on the type of system used, some of them will not be used or can be used optionally.

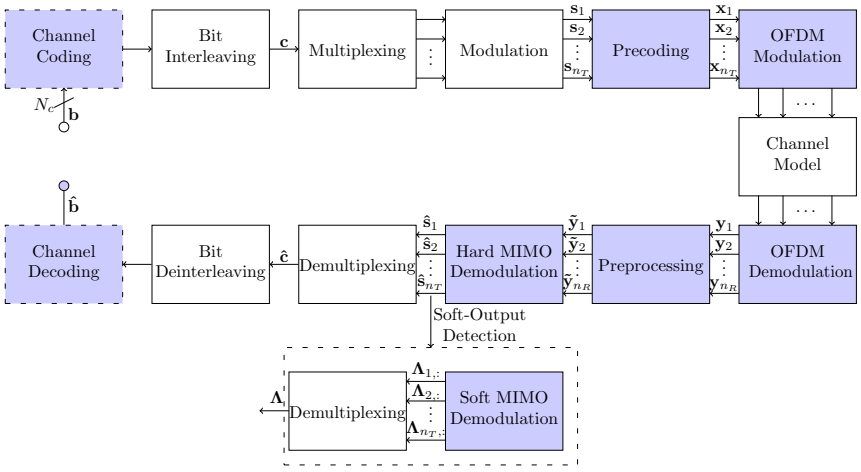


Figure 2.5. High level MIMO system block diagram.

2.3 MIMO Detection

If nearly optimal data detection is desired, the detector becomes often the most computationally expensive part within a MIMO receiver. The demodulator, also known as detector, is responsible of processing the received mixture of signals affected by the channel in order to recover the transmitted data with the accuracy required by the considered application. This issue has motivated the search for high-throughput MIMO demodulators capable to be reconfigured and scalable with the system parameters.

As shown in Figure 2.5, there are two types of detectors depending on the detector output: hard-output detectors, which decide whether a bit is zero or one, and soft-output detectors, which decide the probability (LLR) that a particular bit takes the value zero or one. Next, depending of the performance achieved we can differentiate between optimal, also known as *Maximum-Likehood* (ML) or suboptimal (non-ML) methods. The latter category gets worse performance but has lower complexity. Figure 2.6 illustrates the classification of the most important of the existing MIMO detection strategies. The last classification depends on the detection strategy, which can be either exhaustive, linear, with successive interference cancellation (SIC) way or via a tree search.

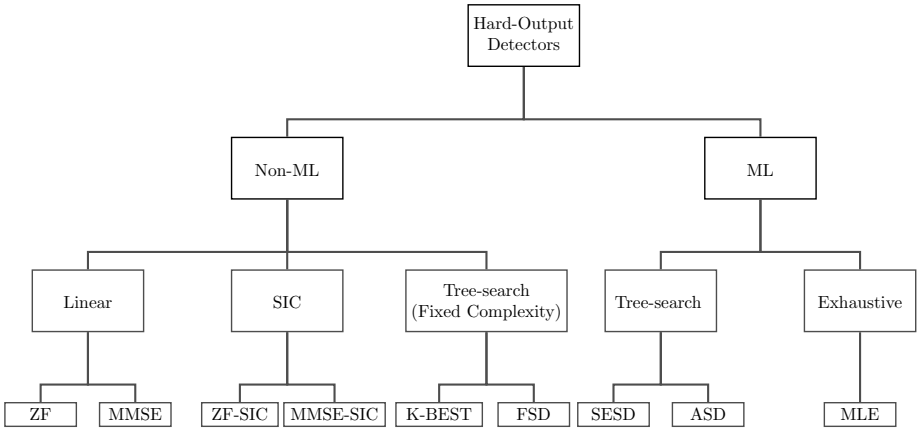
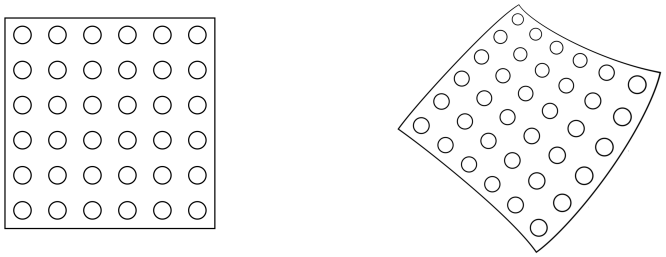


Figure 2.6. Classification of Hard-Output detection algorithms.

2.3.1 Hard-Output Detection

The detection problem in the system model of Fig. 2.2 is usually described in terms of *lattices*. If constellation elements are equally spaced, the set $\mathcal{O}^{n_T}$ forms a rectangular lattice as shown in Fig. 2.7 (a). When these elements are multiplied by the channel matrix $\mathbf{H}$, the lattice suffers a deformation as shown in 2.7 (b). Then, the detection is equivalent to finding the closest point ($\hat{\mathbf{s}}$) to a given point ($\mathbf{y}$) in a lattice generated by the matrix $\mathbf{H}$.

Given the received symbol vector $\mathbf{y}$, the detection problem consists in determining the vector $\hat{\mathbf{s}}$ with the highest a posteriori probability of having been transmitted:



(a) (b)

Figure 2.7. (a) Initial lattice points (b) deformed lattice.

$$\hat{\mathbf{s}} = \arg \max_{\mathbf{s} \in \mathcal{O}^{n_T}} P\{\hat{\mathbf{s}} = \mathbf{s} | \mathbf{y}\}. \tag{2.7}$$

The *maximum-a-posteriori* probability (MAP) rule is obtained applying the Bayes theorem as:

$$\hat{\mathbf{s}} = \arg \max_{\mathbf{s} \in \mathcal{O}^{n_T}} \frac{P\{\mathbf{y} | \hat{\mathbf{s}} = \mathbf{s}\} P\{\mathbf{s}\}}{P\{\mathbf{y}\}}. \tag{2.8}$$

If we assume that $P\{\mathbf{s}\}$ is constant, i.e equally likely, the MAP rule turns into the *maximum-likehood* (ML) detection rule as follows:

$$\hat{\mathbf{s}} = \arg \max_{\mathbf{s} \in \mathcal{O}^{n_T}} P\{\mathbf{y} | \hat{\mathbf{s}} = \mathbf{s}\}. \tag{2.9}$$

Considering an additive, white and Gaussian noise (AWGN), the ML detection can be done by solving the following least squares problem [26]

$$\hat{\mathbf{s}} = \arg \min_{\mathbf{s} \in \mathcal{O}^{n_T}} \|\mathbf{y} - \mathbf{H}\mathbf{s}\|^2, \tag{2.10}$$

where $\| \cdot \|$ denotes the 2-norm or Euclidean Distance (ED) and $\mathcal{O}^{n_T}$ represents all possible transmit symbol vectors [27].

A brute force algorithm, called *Maximum-Likehood Exhaustive* (MLE) detector can directly solve this problem by an exhaustive search over the set of n_T-dimensional lattice points $\mathbf{s} \in \mathcal{O}^{n_T}$. Problem (2.10) is referred in

integer optimization as integer least squares and in lattice theory as *Closest Vector Problem* (CVP) and is known to be (NP)-hard [28].

The size of the search space $|\mathcal{O}|^{n_T} = 2^{m \cdot n_T}$ increases exponentially with the number of transmitter antennas and the number of bits per symbol. Then, its straight implementation based on exhaustive search has a prohibitive complexity, which precludes its use in practical systems (especially for high-order constellations and high number of transmitter antennas). For this reason, more efficient detection schemes were devised and are being used in practice, such as linear detectors or sphere decoding detectors. The most efficient and relevant contributions in hard-output detection research are detailed below.

Linear Detectors

The *Zero Forcing* (ZF) detector is a linear and simple technique for recovery the transmitted signals at the receiver [29]. The estimated transmitted vector can be obtained by means the pseudo-inverse of the channel matrix as follows:

$$\hat{\mathbf{s}}_{\mathbf{ZF}} = \mathcal{Q}\left\{\mathbf{H}^{\dagger}\mathbf{y}\right\} = \mathcal{Q}\left\{(\mathbf{H}^{H}\mathbf{H})^{-1}\mathbf{H}^{H}\mathbf{y}\right\}, \tag{2.11}$$

where $(\cdot)^{\dagger}$ stands for the pseudo-inverse matrix, $(\cdot)^{-1}$ indicates simple matrix inversion and the function $\mathcal{Q}(\cdot)$ assigns the closest constellation symbol which is known as *quantization*. Unfortunately the algorithm gets good performance only when $\mathbf{H}$ is well-conditioned, because after the product by the inverse channel the noise variance can be significantly amplified. A meaningful extension of ZF detector appeared with the aim of counteracting the noise enhancement problem. This method, called *Minimum Mean Square-Error* (MMSE), minimizes the error due to the noise and the interference by using $(\mathbf{H}^{H}\mathbf{H} + \sigma_w^2\mathbf{I})^{-1}\mathbf{H}^{H}$ in (4.4) instead of the pseudo-inverse [29].

Additionally, an iterative extension was proposed in order to improve the performance of ZF method. It is referred as zero forcing with *successive interference cancellation* (SIC) or with *decision feedback* (DF) [30]. In this case, the quantization is done for each component of $\hat{\mathbf{s}}$ successively and not jointly as in the ZF. Note that SIC technique can be applied also to MMSE detector.

Nulling and cancellation detectors have the drawback of error prop-

agation to the next symbols to be detected when a wrong decision has already carried out. It can be shown that it is advantageous to perform the detection following a certain order, which can be sometimes different from the initial one. A quite simple reordering method calculates the norm of all the channel matrix columns and reorders them in ascending order. This is an approach to detect first the symbols with the highest SNR, i.e., the most reliable ones. In [31], an optimal ordering was proposed by the BLAST laboratories (VBLAST) and employed for nulling and cancellation detection. The resulting scheme was named ordered SIC (OSIC) detector.

Tree-Search-Based and Sphere Decoding Detectors

As previously mentioned, the MLE detector can not be used in most situations, especially when high-order constellations and/or large number of transmitted antennas are used. *Sphere Decoding* (SD) methods arose aiming to reduce the range of search, seeking uniquely those lattice points that lie in a hypersphere of a given radius [32], called *sphere radius* (r), around the received vector $\mathbf{y}$ (see Fig. 2.8), i.e. a subset of the total $\mathcal{O}^{n_T}$ possible values of $\mathbf{s}$. The SD search can be expressed introducing this constraint in (2.10) as follows:

$$\hat{\mathbf{s}} = \arg \min_{\mathbf{s} \in \mathcal{O}^{n_T}} \left\{ \|\mathbf{y} - \mathbf{H}\mathbf{s}\|^2 \leq r \right\}. \tag{2.12}$$

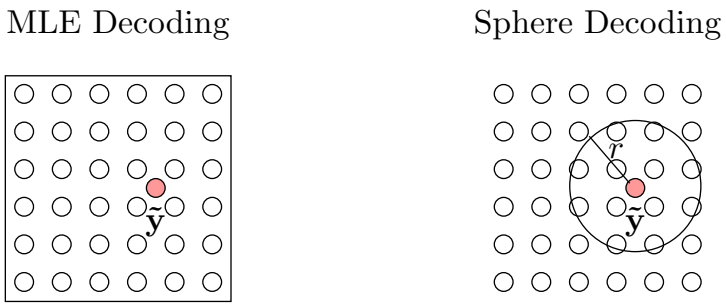


Figure 2.8. Idea behind the Sphere Decoder: the search process is performed only within a sphere of radius r.

A first step to decrease slightly the complexity is to reduce the channel matrix to a canonical form by unitary transformations, being the most

commonly used the QR decomposition. Then, the channel matrix is decomposed into $\mathbf{H} = \mathbf{Q}\begin{bmatrix}\mathbf{R}\\ \mathbf{0}\end{bmatrix}$ where $\mathbf{R} \in \mathbb{C}^{n_T \times n_T}$ is an upper triangular matrix, and $\mathbf{Q} \in \mathbb{C}^{n_T \times n_T}$ is unitary ($\mathbf{Q}^H\mathbf{Q} = \mathbf{Q}\mathbf{Q}^H = \mathbf{I}$). Left-multiplying (2.10) by $\mathbf{Q^H}$ (recall that unitary matrices don't modify the 2-norm of a vector) and calling $\tilde{\mathbf{y}} = \mathbf{Q}^H\mathbf{y}$, the problem 2.10 can be rewritten as:

$$\hat{\mathbf{s}} = \arg \min_{\mathbf{s} \in \mathcal{O}^{n_T}} \|\tilde{\mathbf{y}} - \mathbf{R}\mathbf{s}\|^2, \tag{2.13}$$

where the most likely transmitted symbol vector $\hat{\mathbf{s}}$ is found by searching the smallest Euclidean Distance (ED) between the received vector $\tilde{\mathbf{y}}$ and each possible vector $\mathbf{s}$.

To clarify how the triangular structure of $\mathbf{R}$ can be exploited, Eq. 2.13 has been expressed in a more explicit way as

$$\hat{\mathbf{s}} = \arg \min_{\mathbf{s} \in \mathcal{O}^{n_T}} \left\{ \sum_{i=n_T}^{1} \left| \tilde{y}_i - \sum_{j=i}^{n_T} R_{i,j} s_j \right|^2 \leq r \right\}. \tag{2.14}$$

Problem (2.14) can be represented as a decision tree with $n_T + 1$ layers, M branches emerging from each non-leaf node, and M^{n_T} leaf nodes. The tree-paths are built by connecting nodes and stand for candidate solutions. For instance, a tree-path containing selected symbols from the root up to level 1 has the form

$$s = [s_1, s_2, \ldots, s_{n_T}]^T. \tag{2.15}$$

Non-negative weights δ_i and η_i are associated with the branches and nodes, respectively (see Fig. 2.9). We associate the *partial Euclidean distance* (δ) metric ("*branch weight*") to any branch as follows:

$$\delta_i = \left| \tilde{y}_i - \sum_{j=i}^{n_T} R_{i,j} s_j \right|^2. \tag{2.16}$$

Also, we associate a *cumulative Euclidean distance* (η) metric to any node as:

$$\eta_i = \eta_{i+1} + \delta_i, \qquad i = n_T, n_T - 1, \ldots, 1 \tag{2.17}$$

where the initial distance for the root node η_{n_T+1} is set to 0.

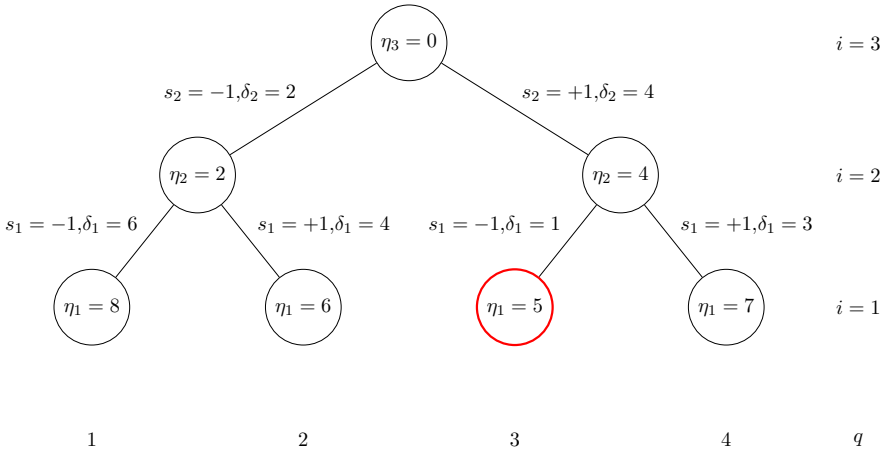


Figure 2.9. Decoding search tree for a 2×2 MIMO system using a BPSK constellation.

Then, SD decoding descends through the tree-paths at each i-th level and computes η_i such as (2.17) until (2.14) is not fulfilled. When the cumulated metric of a node is higher than the radius, the remaining subtrees can be pruned, accelerating the search process. The survivors paths (vectors inside the hypersphere) will form part of the candidate points to be the ML solution. Many different tree-search strategies have been proposed during the last years, some of which can be found in [33][34][35].

The complexity of tree-search detectors is commonly measured in number of expanded nodes, since this allows a fair comparison among different algorithms. The selection of a suitable radio is very important in order to find the ML solution visiting the fewest possible number of nodes. Several methods can estimate the sphere radius [32][36]. It is often used as initial radius the distance provided by a suboptimal and low-complexity detection algorithm such as Zero Forcing (ZF) solution $\hat{\mathbf{s}}_{\mathbf{ZF}}$ as:

$$r = \|\mathbf{y} - \mathbf{H}\hat{\mathbf{s}}_{\mathrm{ZF}}\|^2. \tag{2.18}$$

Thus, choosing properly r is very important in the SD detection performance. For example, if we choose a too small radius, there may not be candidates within the hypersphere. However, if we choose a too large radius the complexity of the algorithm can be nearly the same as the MLE, since there will be many points within the hypersphere.

In order to avoid the problem of the estimation of the suitable initial radius, different methods were devised such as *Fincke-Pohst Sphere Decoder* [33] and *Schnorr Euchner Sphere Decoder* (SESD) [34] and the *Automatic Sphere Decoder* (ASD) [35]. The SESD performs a search from the top to the bottom of the tree with radius reduction. First, the algorithm initializes the sphere radius with $r = \infty$, and is updated any time a leaf node is visited. Every time a leaf node is reached with cumulative metric $\eta_1 < r$, the detector updates the radius as $r = \eta_1$ and continues exploring tree branches with a smaller sphere radius.

The ASD detector does not need an initial sphere radius to find the ML solution. For this purpose, a list of nodes is kept during the exploration of the tree, which contains and defines the bound between the explored part and the unexplored parts of the tree. The ASD starts initializing the list with the root node, to which an accumulated ED is associated equal to zero. In each iteration, the ASD selects and expands the node inside the list with the smallest cumulative Euclidean distance (η). Then, the expanded node is replaced by its M children nodes and removed from the list. The same process continues until a leaf node is reached. This method guarantees that the first node visited in the last level will be the node that contains the optimal (ML) solution. Some variants of ASD have been developed to speed up the detection limiting the number of expanded nodes, such as using a constraint radius [37].

Additionally, some variants of the SD were devised in order to decrease the computation required to detect a signal such as box optimization [38].

Fixed-Complexity Detectors

The variable complexity of Sphere Decoding methods is a critical issue regarding their practical use in a MIMO system. In the SD algorithms the number of visited nodes can not be known in advance because it depends on the chosen radio, the noise and channel conditions. For this reason several methods were proposed in order to address these problems. For example, the *K-Best Sphere Decoder* (K-BEST) [23] expands the detection

tree from top to bottom and considers only those K survivor candidates that show the smallest cumulative Euclidean distance η_i at each level i of the tree. Then, the search continues from these K survivor paths and follows the same strategy until the lowest level is reached. The detected signal vector $\hat{\mathbf{s}}$ is given by the path from the root up to the leaf node with the smallest Euclidean distance. The main advantage of this method is that the maximum number of expanded nodes is limited by K and can be known in advance.

Similarly, a method called *Fixed-Complexity Sphere Decoder* (FSD) was proposed in order to fix the complexity of the Sphere Decoder [39]. The FSD algorithm combines a preprocessing (sorting) stage with a tree-search composed of two stages:

- Full Expansion (FE): In the first n_E levels, all the possible values of the constellation for each survivor path are assigned to the symbol at the current level.
- Single-Path Expansion (SE): The SE stage starts from each retained path and obtains the remaining unknow symbols (those in the lowest $n_T - n_E$ tree-levels) using SIC detection (see section 2.3.1).

The preprocessing stage sorts the columns of the channel matrix $\mathbf{H}$ in order to place the symbols that suffer the largest noise amplification in the first n_E levels since no candidate is being discarded at this level and the final performance will not be altered. Then, accordingly, the symbols that suffer the smallest noise amplification are placed at SE levels.

These methods have been implemented over different hardware technologies such as FPGA and GPUs [40], particularly FSD algorithm because its structure is well suited for parallel computing [41][42][43].

2.3.2 Soft-Output Detection

Assuming that all transmit vectors are equally likely, the *optimal soft MAP* (OMAP) *demodulator* calculates the exact LLR for $c_{i,k}$ as

$$\Lambda_{i,k} = \log \frac{\mathrm{P}(c_{i,k}=1|\mathbf{y},\mathbf{H})}{\mathrm{P}(c_{i,k}=0|\mathbf{y},\mathbf{H})} = \log \frac{\sum_{\mathbf{s}:s_i\in\mathcal{O}_k^1} e^{-\frac{\|\tilde{\mathbf{y}}-\mathbf{R}\mathbf{s}\|^2}{\sigma_w^2}}}{\sum_{\mathbf{s}:s_i\in\mathcal{O}_k^0} e^{-\frac{\|\tilde{\mathbf{y}}-\mathbf{R}\mathbf{s}\|^2}{\sigma_w^2}}}, \tag{2.19}$$

where $\mathcal{O}_k^u$ denotes the set of all symbols $\mathbf{s} \in \mathcal{O}$ such that the bit in position k has the value u. The complexity of this method is $\Theta(|\mathcal{O}|^{n_T})$ since the LLRs are calculated for all layers n_T, therefore is mandatory the computation of $|\mathcal{O}|^{n_T}$ distances.

If the receiver uses a *max-log approximation* (MLA) demodulation the computation of the LLRs for each code bit is calculated according to [44]

$$\Lambda_{i,k} \approx \frac{1}{\sigma_w^2}\left[\min_{\mathbf{s}:s_i\in\mathcal{O}_k^0} \|\mathbf{y}-\mathbf{H}\mathbf{s}\|^2 - \min_{\mathbf{s}:s_i\in\mathcal{O}_k^1} \|\mathbf{y}-\mathbf{H}\mathbf{s}\|^2\right]. \tag{2.20}$$

It is useful to realize that the Euclidean distance of the solution $\hat{\mathbf{s}}^{\mathrm{HARD}}$ of the hard-output detection problem (2.13) directly provides one of the two minima in expression (2.20), denoted in what follows as η^{HARD}:

$$\hat{\mathbf{s}}^{\mathrm{HARD}} = \arg\min_{\mathbf{s}\in\mathcal{S}} \|\mathbf{y}-\mathbf{H}\mathbf{s}\|^2, \qquad \eta^{\mathrm{HARD}} = \left\|\mathbf{y}-\mathbf{H}\hat{\mathbf{s}}^{\mathrm{HARD}}\right\|^2, \tag{2.21}$$

where $\mathbf{s} \in \mathcal{S}$ denotes the subset of the whole transmit constellation considered by the hard-output detection. The second minimum in (2.20) for each $i = 1, \ldots, n_T$ and $k = 1, \ldots, m$, can be computed as

$$\bar{\eta}_{i,k} = \min_{\mathbf{s}:s_i\epsilon\mathcal{O}_k^{[s_i^{\mathrm{HARD}}]_{\bar{k}}}} \|\mathbf{y}-\mathbf{H}\mathbf{s}\|^2, \tag{2.22}$$

where $[\cdot]$ indicates the binary-valued representation of the constellation symbol and $[s_i^{\mathrm{HARD}}]_{\bar{k}}$ denotes the bit-negation of bit $[s_i^{\mathrm{HARD}}]_k$.

Then, the LLRs can be calculated as

$$\Lambda_{i,k} = \frac{1}{\sigma_w^2}(\eta^{\mathrm{HARD}} - \bar{\eta}_{i,k})(1 - 2[\hat{s}_i^{\mathrm{HARD}}]_k). \tag{2.23}$$

There are numerous suboptimal alternatives of soft MIMO detectors designed to avoid an exhaustive search over the entire range of possibilities $|\mathcal{O}|^{n_T}$. In fact, any hard detector can produce soft values via the max-log approximation [45][46].

2.4 High Performance Simulation Libraries

High Performance Computing (HPC) systems allow to reduce the execution time of complex problems, but it can lead to serious programming difficulties. The programming challenge involves the developers to know in depth different programming languages and the features of the architecture. In this sense, high performance libraries become valuable tools for specialists of a particular field, since they ease the development of scientific codes.

Some software companies have already released to the market various HPC libraries; these libraries not only facilitate the preparation of code but also exploit the huge computing capabilities of new architectures to accelerate and optimize these implementations. In the field of communication systems we can find two remarkable libraries which are described in the following sections.

2.4.1 Simulation Acceleration using MATLAB

The *Communications System Toolbox* (CST) allows the acceleration of MIMO system simulation using GPUs, multi-core and code generation [47]. This toolbox provides algorithms for designing, simulating, and analyzing communications systems. These capabilities are provided as MATLAB functions, MATLAB System objects, and Simulink blocks. The toolbox enables source coding, channel coding, interleaving, modulation, equalization, synchronization, and channel modeling. Also it allows to analyze bit error rates, generate eye and constellation diagrams, and visualize channel characteristics. Furthermore, dynamic communications systems that use OFDM, OFDMA, and MIMO techniques can be modelled. Algorithms support fixed-point data arithmetic and C or HDL code generation. If the user has installed the *Parallel Computing Toolbox* (PCT) CST enables to accelerate the simulation using multi-core and GPU hardware within the computer [48]. Although this software is excellent and widely used by the scientific community, at present just a few functions are prepared to use parallel computing with GPUs. The list of functions with GPU support is:

- LDPC Decoder
- Turbo Decoder
- Viterbi Decoder
- Convolutional Encoder
- PSK Demodulator
- PSK Modulator

- Block Deinterleaver
- Block Interleaver
- Convolutional Deinterleaver
- Convolutional Interleaver
- AWGN Channel

2.4.2 Simulation Acceleration using IT++

IT++ library is a C++ library of mathematical, signal processing and communication classes and functions [19]. Its main use is simulation of communication systems and performing research in the area of communications. The kernel of the library consists of generic vector and matrix classes, and a set of accompanying routines. Such a kernel makes IT++ similar to MATLAB, GNU Octave [49] or SciPy [50]. IT++ makes an extensive use of existing open-source or commercial libraries for increased functionality, speed and accuracy. In particular BLAS, LAPACK and FFTW libraries can be used. Instead of the reference BLAS and LAPACK implementations, some optimized platform-specific libraries can be used as well, i.e. Automatically Tuned Linear Algebra Software (ATLAS) [51], Intel Math Kernel Library (MKL) [52] and AMD Core Math Library (ACML) [53]. However, this library is oriented to its exclusive use on multi-core machines; nowadays it does not support GPUs.

2.4.3 MIMO Design using LabVIEW

The LabVIEW simulation package from National Instrument shows a graphical and fast simulator in order to model the MIMO-OFDM system [54]. By using this simulator, the bit errror rate (BER) performance of the system can be obtained. The list of functions provided by this simulator is:

- Pilot Sequence Generator
- Random Bit Generator
- Symbol mapper/demapper
- Pilot-to-data Power Ratio (PDPR)
- Subcarrier Allocator
- MIMO-OFDM Transmitter
- MMSE MIMO-OFDM Receiver
- MMSE Channel Estimator
- 2D-DFT Matrix Generator

There are some other LabVIEW simulation toolkits for MIMO and

OFDM systems such as the MIMO toolkit developed by Prof. Heath's research group, which offers a simple tool to simulate MIMO systems [55].

2.5 Conclusion

In this chapter, firstly different transmission schemes derived from the well-known MIMO BLAST system have been described: the MIMO-BICM system where the soft demodulation is considered and the MU-MIMO system, which needs precoding techniques in order to reduce the detection complexity at the receiver side. For the sake of completeness, a general block diagram of MIMO simulation was presented with the most complex and important functional/computational blocks.

Different methods and implementations available in the literature to process the detection, preprocessing and precoding blocks have been reviewed. First, different hard and soft output detection methods were revised. Detection techniques can be classified according to their performance in terms of Bit Error Rate between optimal and suboptimal methods; generally being the most precise those with a higher computational cost. Detection techniques can be classified depending on the detection strategy, which can be either exhaustive, linear, with successive interference cancellation (SIC) way or via a tree search. Next, some preprocessing techniques that were proposed for decreassing the computational complexity of the optimal detectors and also to increase the accuracy of the suboptimal ones were briefly introduced.

In the last part of the chapter, have been presented some software toolkits devised to simulate transmissions in a MIMO system. Each tool is primarily designed to be used from different programming environments: MATLAB, C++ and LabVIEW. Furthermore, some of them have support for working with GPU or multi-core and also rely on the use of other highly optimized numerical linear algebra libraries. However, it does not exist a tool that possesses all the mentioned features nor meets target user's main needs (see Table 2.1).

In light of the scarce available tools and the possible wastage of resources of the user's machine, it seems clear the need of the design and development of a library able to fully exploit the computational capacity of the parallel computing architectures. In this sense MIMOPack can help

Table 2.1. Comparison of features related to the HPC between different MIMO simulation toolkits.

Description	CST	IT++	MIMOPack
Multi-core Support	•	•	•
GPU Support	•	—	•
Multi-GPU Support	—	—	•
Heterogeneous computing	—	—	•
MATLAB interface	•	—	•
C/C++ interface	—	•	•
3rd HPC libraries support	•	•	—

stakeholders to accelerate those parts of MIMO systems with high computational requirements, developing and deploying computational tools and functions enabling scientists to model, simulate, analyze and predict the communication system behavior. From a functional standpoint, the tools available are very limited, since for each computational block does not always provide different solutions or methods. Therefore, it would be interesting to offer different algorithms for solving the same block, so that the user can decide which one fulfills their needs and the parameters of the MIMO system.

Tools and Optimization Techniques 3

3 Tools and Optimization Techniques

The simulation of MIMO transmission systems entails several stages with high computational complexity. This complexity becomes especially large when the size of different simulation parameters increases: number of signals to simulate, number of transmitter and receiver antennas and number of constellation symbols. But also when the user needs greater reliability in the system performances, which involves the use of: ML detectors, coded transmissions schemes (channel decoders and soft-output detectors).

In order to reduce the computational cost and the simulation time, a set of software and hardware tools that allows the efficient computation of some MIMO systems modules in Fig. 2.5 are introduced. Additionally, a method consisting in the efficient calculation of the Euclidean distances needed to perform the hard-output and soft-output detection is presented in the last part of the chapter. This optimization is used in most of the algorithms and therefore is the key to understand the implementations shown in the following chapters.

3.1 Hardware Tools

Modern High Performance Computing systems present basically two major types of components: multi-core central processing units and hardware accelerators. Multi-core processors and Graphics Processing Units as accelerators have been used throughout this thesis.

3.1.1 Multi-core Processors

During the last years, the main microprocessors manufacturers such as Intel or AMD have been focused in developing faster and smarter chips. From 1983 to 2002 the technology used to improve the performance was based on the increment of the clock frequency. This trend caused a quantum leap in the clock rates from 5 MHz to 3 GHz but at the expense of a high requirement of power, thus complicating the manufacture of this kind of processors.

To address this challenge, additional techniques have been developed to enhance computing capabilities. This technique fits multiple processing units (called cores) onto a single processor. The cores can process simultaneously multiple tasks although at a lower clock rate. The last Intel architecture, codename Broadwell, uses 14 nm technology in processors with 2, 4 and 18 cores. On the other hand, AMD developed the Bulldozer architecture for desktop environments, with 32 nm technology and 4, 6 and 8 cores, and the Opteron series for servers, with 8 and 12 cores. The purpose is to get optimal performance with minimal consumption in all cases. However, new objectives of both companies have appeared due to the GPU growing market in recent years. Intel started the Larrabee architecture GPGPU project some years ago with an uncertain future. After eight years of development, Intel has announced its Xeon Phi Coprocessor that is an 60-core processor with a GPU-like conception [56]. Although the performance results reported by the company show that the Xeon Phi achieves more than 1 teraFLOPS of double-precision and more than 2 teraFLOPS of single-precision, it seems that it can get this performance only under very specific conditions. On the other hand AMD decided to design an Accelerated Processing Unit (APU) called Heterogeneous System Architecture (HSA) that combines features of the CPU, GPU and other processing elements on a single chip [57].

3.1.2 Graphics Processing Units

A Graphics Processing Unit (GPU) is a coprocessor originally designed for accelerating the computation of computer graphics. However, in recent years a trend in the scientific computing community has appeared based on the use of the GPU to handle general purpose problems traditionally solved by the Central Processing Unit (CPU), called General-Purpose Computing on Graphics Processing Units (GP-GPU).

GPUs are fascinating tools and represent a quantitative leap in the development of high performance hardware. Nowadays, they are present in almost all computing systems, laptops, PCs and supercomputers [58]. The fast advance in programmability has allowed its use to deal with many problems with an insatiable appetite for computing power. This kind of problems can appear in fields as different as Climate Science [59], Biochemical [60], or Signal Processing [61]. Probably it would be impossible to go back, and the near future cannot be imagined without GPU technology.

GPUs exhibit a fast evolution. The last models on the market have more cores, more computational power and several new features [62][63]. Software tools also allow a friendlier GPU programming than some years ago: CUDA (Compute Unified Device Architecture, [64]) is continuously evolving and the most recent SDK versions solve problems of previous versions; OpenCL [65] language seems a good future alternative but unfortunately it does not show the evolution speed and the performance of CUDA.

The new NVIDIA Kepler architecture is designed to maximize computational performance with superior power efficiency, by offering new features to optimize and increase parallel workload execution and therefore offering much higher processing power than the previous Fermi architecture. The new architecture shows relevant improvements that make hybrid computing easier (e.g dynamic parallelism), and are suitable for a wider set of problems (see [62] for a detailed description). The last models by NVIDIA, codename Maxwell, have kept the increase in computational power by using a more sofisticated hardware with up to 5000 cores into the chip [66].

Kepler is the most modern architecture that has been used to test the performance of the present library, for this reason the Kepler computing architecture has been selected here to describe the GPU computing architecture. Figure 3.1 shows the GK110 Nvidia Kepler composed by 15 next generation streaming multiprocessors (SMX), each SMX including 192

CUDA cores, yielding a total of 2880 cores.

Figure 3.1. NVIDIA Kepler GK110 architecture.

The new SMX introduces new special features; as show in Fig. 3.2 each SMX has four warp schedulers and two instruction dispatcher units allowing the concurrent execution of two independent instructions per warp. A warp is a group of 32 parallel threads. Each core has fully pipelined floating-point and integer arithmetic logic units. Furthermore, each SMX has 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

GPU's memory hierarchy is organized as shown in Fig.3.3. There exist different types of memories whithin the GPU, the first one is the global memory (DRAM), which is located outside the SMX (off-chip). This is the slowest memory and limits the application performance in most cases. The global memory is connected with the CPU through a PCI Express bus. The remaining memories (on-chip) provide high-bandwidth access to the data and are briefly described below:

- Configurable shared and L1 cache: This memory can be split as 48kB/16kB, 32kB/32kB or 16kB/48kB between shared memory and L1 cache.
- Read-only cache: Is a 48KB read-only cache.
- L2 cache: The Kepler GK110 has duplicated the amount of L2 cache of the previous Fermi architecture [67]. It avoids the global memory

Figure 3.2. SMX architecture of GPU NVIDIA GK110 Kepler.

bandwidth bottleneck providing high speed data sharing among the GPU.

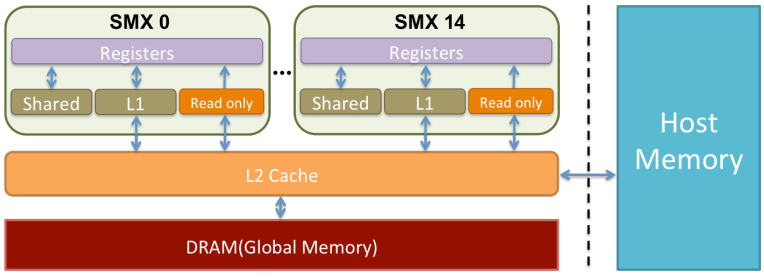


Figure 3.3. Memory hierarchy of GPU NVIDIA GK110 Kepler.

3.1.3 Computer System for simulation testing

In order to assess the performance and the portability of MIMOPack, several test and measurements have been carried out over a Computing System (CS). This system consists of a set of multi-processors and several NVIDIA GPUs. Some features and specifications are listed below:

Computer System A

- CPU: Two Intel Xeon CPU E5-2697 at 2.70 GHz.
- CPU cores: 12 cores per CPU
- Hyperthreading: Yes
- Operative System: Linux CentOS release 6.5
- Architecture: x86_64
- GPU: Two Tesla K20Xm
- Compute Capability: 3.5
- CUDA SDK: 5.5
- Architecture: Kepler
- Number CUDA of cores: 14 (SM) x 192 cores = 2688 cores.

3.2 Software Tools

In this section the most popular linear algebra libraries and the frameworks used to implement the library are described: OpenMP for multi-core programming, CUDA to program NVIDIA GPUs and MEX-Files to execute programs written in C, C++ and CUDA programming languages from MATLAB.

3.2.1 OpenMP Programming Model

OpenMP is an Application Programming Interface (API) [68] for programming shared-memory parallel computers. It consists of a set of compiler directives, callable library routines and environment variables that may be embedded in a code written in a programming language such as Fortran or C/C++ and operating systems such as GNU/Linux, Mac OS X, and Windows.

A master thread launchs a number of slave threads and divides the workload among them (see Fig.3.4). The runtime will attempt to allocate the threads to different processors and the threads will run concurrently.

Here, we describe two of the basic pragmas used in the implementation of some library functions.

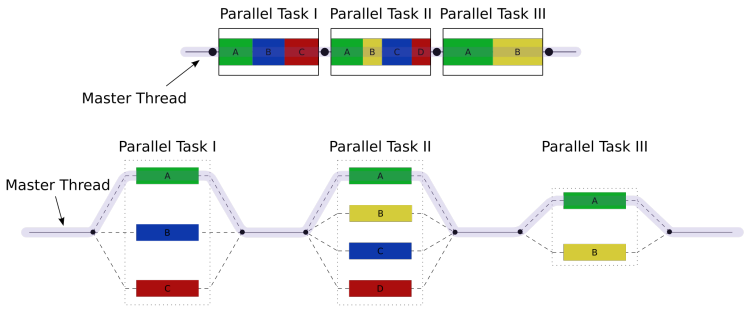


Figure 3.4. An illustration of multithreading where the master thread forks off a number of threads which execute blocks of code in parallel.

- **Parallel for**: it distributes the for loop iterations among the threads. Syntax:

```
#pragma omp parallel for [clause[[,] clause] ...]
<for_loop>
```

- **Parallel sections**: it assigns consecutive but independent code blocks to different threads. Syntax:

```
#pragma omp parallel sections [clause[[,] clause] ...]
{
    [#pragma omp section]
        statement-block
    [#pragma omp section]
        statement-block
    .
    .
    .
}
```

3.2.2 CUDA Programming Model

Compute Unified Device Architecture (CUDA) [64] is a set of programming tools developed by NVIDIA that allows to execute programs written with C/C++, OpenGL, Fortran, and other languages on GPUs.

The GPUs follow a single-instruction multiple-data (SIMD) programming model, that is, a single set of instructions is executed on different data sets. In this model, the programmer defines the kernel function which contains a set of common operations. At runtime, the *kernel* is called from the main central processing unit (CPU) and spawns a large number of threads blocks, which is called *grid*. Each thread block contains multiple threads, usually up to 1024, and all the blocks within a grid must have the same size. Blocks and grids can be one-dimensional, two-dimensional or tri-dimensional but they must not exceed a certain size stated in the GPU's specifications. Each thread can select a set of data using its own unique ID and execute the kernel function on the selected set of data independently. Threads of the same block can share data between them by using the *shared memory* type. However, threads of different blocks are independet and should use *global memory* to share data once all threads have finished running the full kernel.

Figure 3.5 shows a simple example of parallel programming with CUDA. It contains a parallel implementation of a sum of two n-dimensional vectors x and y, the output is stored in a third vector z. The kernel is denoted by the $\_\_global\_\_$ keyword and computes each element of vector z in parallel. The call syntax $sum <<< block, threads >>> (\ldots)$ launches the kernel. Each thread determines the element to process by using its unique ID using: its block index ($blockIdx.x$), its thread index ($threadIdx.x$) and the number of threads per block ($blockDim.x$).

```
__global__ void sum(double *x, double *y, double *z, int n){
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    if(id < n)
        z[id] = x[id] + y[id];
}

void cuda_example(double *x, double *y, double *z, int n){
    // Launch sum kernel with blocks of 64 threads
    sum<<<ceil(n/64), 64>>>(x, y, z, n);
}
```

Figure 3.5. Example of CUDA programming: addition of two input vectors (x and y) of lenght n and outputs a vector of lenght n (z).

3.2.3 MATLAB MEX-Functions

MATLAB is an excellent development tool widely used in various scientific fields [69]. Matlab stands out for its simplicity and efficiency for programming of algorithms based on matrix operations. However, its performance is not as good as expected for non matrix-based problems.

The use of MEX-files allows to call codes written in C/C++ or Fortran from MATLAB without having to rewrite them as M-files [70]. This allows developers to increase the speed of those parts of the code whose structure does not offer good performance in MATLAB. Furthermore, it allows to run MEX-Functions including CUDA code.

Figure 3.6 shows a simple MEX-File example to call the CUDA function seen in the previous section. The name of the gateway routine must be **mexFunction** and acts as an interface between C and MATLAB. This function takes two input matrices x and y which can be accessed using the array of pointers *prhs* (parameters of right hand side). These arguments are collected using *mxGetPr* function. The size of the input matrices are obtained using *mxGetM* and *mxGetN* that return the number of rows and columns, respectively. The output argument is a matrix 1-by-n called z, a matrix of these dimensions is created using *mxCreateDoubleMatrix*. Finally we indicate that this matrix will be the output of the algorithm through the *plhs[0]* pointer (parameters of left hand side).

3.2.4 HPC Linear Algebra Libraries

High Performance Computing linear algebra libraries are commonly used in the development of efficient linear algebra software due to its excellent performance. These packages have been designed to solve big matrix-based problems. However, the simulation of communications systems shows two serious characteristics that hinder its use in MIMOPack library:

- The problems that appear in most stages MIMO communications seen in Figure 2.5 are not matrix-based problems.

- MIMO channel matrices are very small, usually 2×2 or 4×4. Even if we consider very large arrays (e.g 100×100) are not large enough to obtain significant performance improvements.

```
/*
 * arraySum.c - example in MATLAB External Interfaces
 * Performs the addition of two input 1xN matrices (A and B)
 * and outputs a 1xN matrix (C)
 *
 * The calling syntax is:
 * C = arraySum(A, B)
 * This is a MEX-file for MATLAB.
*/

#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
mxArray *prhs[]){

  double *x, *y, *z;
  int m, n, p;

  x = mxGetPr(prhs[0]);
  y = mxGetPr(prhs[1]);
  n = mxGetN(prhs[1]);

  plhs[0] = mxCreateDoubleMatrix(1, n, mxREAL);
  z = mxGetPr(plhs[0]);
  cuda_example(x, y, z, n);
}
```

Figure 3.6. Example in MATLAB External Interfaces: performs the addition of two input $1 \times N$ matrices (x and y) and outputs a $1 \times N$ matrix (z).

BLAS (Basic Linear Algebra Subprograms)

The BLAS (Basic Linear Algebra Subprograms) is a set of routines for performing basic vector and matrix operations [71]. These routines are classified in different levels based on the type and complexity of operations: functions of level 1 perform scalar, vector and vector-vector operations, level 2 perform matrix-vector operations, and the level 3 perform matrix-matrix operations.

LAPACK (Linear Algebra Package)

The LAPACK (Linear Algebra Package) contains routines for solving systems of linear equations, linear least squares problems, factoring matrices, eigenvalue problems, and singular value problems over dense matrices [72]. Important companies have made their own implementation optimized for its processors, for example:

- MKL (Math Kernel Library): Optimized for Intel processors [52].
- ACML (AMD Core Math Library): Optimized for AMD processors [73].
- CULA: A GPU-accelerated implementation of LAPACK library [17].
- MAGMA: Efficient implementation of LAPACK with GPU for heterogeneous/hybrid architectures starting with current "Multi-core+GPU" systems [16].

3.3 Heterogenous computation

As seen in section 3.2.2, programming with GPUs entails serious difficulties, especially if they are intended to be used as general purpose machines. Sometimes these difficulties are intrinsic and related to the SIMD (Single Instruction Multiple Data) model, which essentially allows data-parallelism but limits the use of graphics accelerators in general purpose applications. Instead, MIMD (Multiple Instruction Multiple Data) models allow task parallelism and can be more efficient in this last type of applications.

Some GPU limitations are derived from their technical specifications (for instance, their clock frequency is lower than the current-generation CPU), from usage and memory capacity limitations and, especially, from the fact that GPUs exist as accelerators and not as chips that include all the features of the set CPU-GPU. Therefore it is imporant to identify the problems that can be solved with these tools:

- High computational requirements: The GPUs have a lot of cores which must be utilized upto their full power as much as possible.

Problems with a low workload are disadvantageous for its programming on the GPU, since they consume too much time communicating but not enough time in computation.

- Low communication: Communications between the CPU and the GPU can decrease the throughput of the GPU, especially when the GPU must wait for the results of the CPU. Algorithms where the different threads must synchronize frequently will not profit from the full computing power of the GPU.

One of the more decisive concepts for successfully programming a computer that uses GPU is the underlying model of parallel computer. Traditionally, a GPU card has been considered as an isolated parallel computer, fitting a SIMD model, and connected to a sequential computer (see Fig. 3.7). From this point of view, the GPU card can be seen as a set of processors, running the same instruction simultaneously, each one on its own set of data. An appropriate performance metric for this system could be the speedup achieved by the graphics card against the CPU. This speedup can be obtained by dividing the runtime of a program executed in the sequential processor by the execution time given by the graphic card. A similar metric consists of comparing the Gflop/s needed by the CPU and the GPU to solve the same problem. Note that the different clock speed of operation in the sequential unit and in the GPU causes an unfair comparison between performances of CPU and GPU. In fact, this is not a classical speedup because the considered sequential machine is not an instance of the considered parallel machine for just one processor.

A more realistic model should consider the host system and graphics card as a whole, and the host computer as another parallel computer, at the same level than the GPU. This leads us to the heterogeneous parallel computer model. A similar model is used for instance in [74]. Following this idea, a system with a GPU or an accelerator card (see Fig. 3.8) consists of a set of two (or more) parallel computers, with different speeds, each of them with access to different types of memory, which also implies different memory access times for each processor.

A model of this kind would be characterized by the number and type of processors, and different access time of each processor to the different types of memory. For example a system comprising a multi-core type CPU is considered in Fig. 3.8. This system has a first-level cache and a main

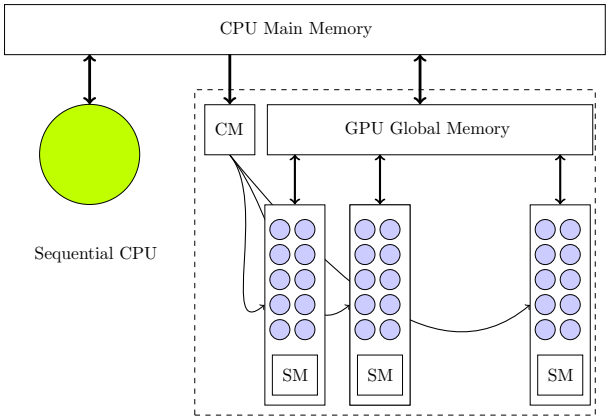


Figure 3.7. Sequential computer with a GPU accelerator.

memory, shared by all cores, and an accelerator many-core type card with different types of memory (global memory, constant memory (CM), shared memory (SM)). In this case the CPU can write and read to global and constant memory of the GPU and GPU can write and read to its global memory and only read from constant memory.

Performance and programming of this model depend on: the type of parallel computer (MIMD in the case of the CPU, SIMD in the case of the GPU), the clock speed of CPU and GPU, the access time to each type of memory and the amount of memory in each memory class. Note that the performance of a GPU in a system of this kind is difficult to evaluate as an isolated component. The best metric in this case may be to compare the speed of the system with and without the accelerator card. It must be allowed (and even encouraged) a simultaneous use of the GPU and CPU, and compare the Gflop/s obtained when they act together and when eliminating the use of the GPU to solve a given problem.

This second approach is much more realistic, and it is used, for example, in the case of numerical linear algebra libraries like MAGMA [16] or CULA [17]. Although these libraries are considered specific libraries for GPU, they run usually part of their programs on the CPU and reserve the GPU to execute those parts that exhibit a strong data parallelism.

MIMOPack library allows the execution of several functions in a heterogeneous mode to fully exploit the computational capacity. The workload

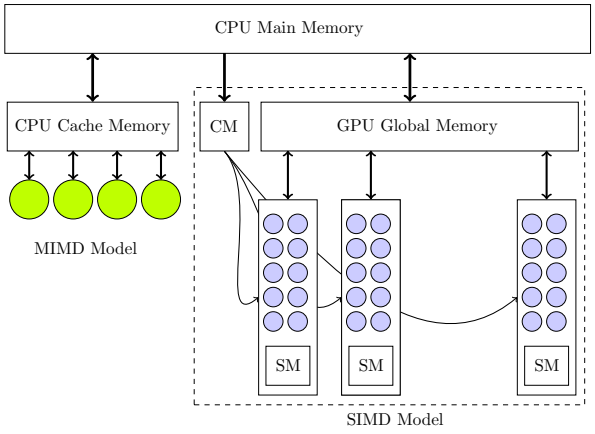


Figure 3.8. Heterogeneous parallel computer system.

can be splitted between the different types of architectures on the target machine (GPU or CPU). Normally, the user will try to simulate a large number of transmitted signals, then the dispatcher splits the amount of signals among different resources.

Figure 3.9 illustrates the detection of N_c signals with several Nvidia GPUs and OpenMP threads. As input of the parallel functions, the user must specify the workload percentage to be computed by the GPUs (p_w). Thus, the number of signals to be estimated by the GPU is calculated as $N_c^{(gpus)} = N_c \cdot \frac{p_w}{100}$ and the number of signals to be estimated by the CPU as $N_c^{(gpus)} = N_c \cdot \frac{100-p_w}{100}$.

The master thread splits the computation according to the selected number of GPUs (n_{gpus}) and openMP threads (n_{cpus}). One of the threads will distribute the computation of $N_c^{(cpus)}$ signals among n_{cpus} threads. The other thread is in charge of controlling and create as many threads as GPUs have been selected and deliver the computation of $N_c^{(gpus)}$ signals among the n_{gpus} GPUs. To carry out this distribution the option "omp_nested" must be previously activated thus allowing nested parallelism.

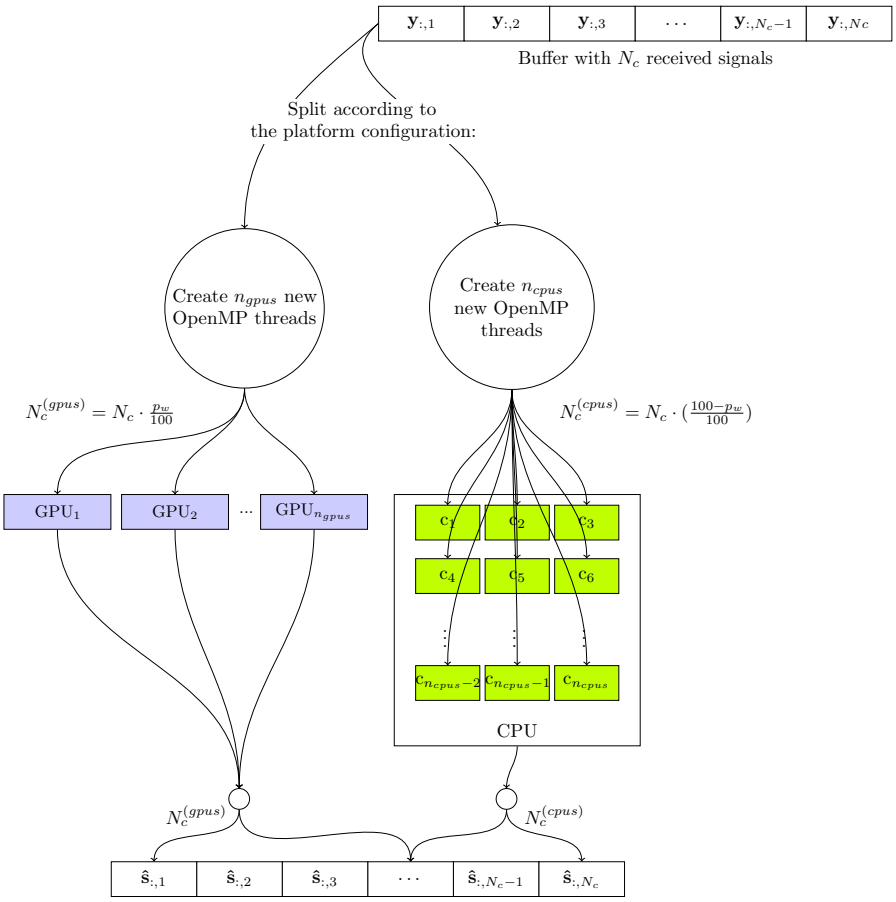


Figure 3.9. MIMOPack workload distribution for the simulation of N_c signals on a heterogeneous parallel computer system.

3.4 Efficient Euclidean Distance Calculation

The *Efficient Euclidean Distance Calculation* (EEDC) optimization is devised in order to reduce the computational complexity of the MIMO detectors. As shown in section 2.3.1, the ML detection problem (2.13) can be expressed in a more explicit way as

$$\hat{\mathbf{s}} = \arg \min_{\mathbf{s} \in \mathcal{O}^{n_T}} \left\{ \sum_{i=n_T}^{1} \left| \tilde{y}_i - \sum_{j=i}^{n_T} R_{i,j} s_j \right|^2 \right\}. \tag{3.1}$$

EEDC consists in calculating all posible values of the inner sumatory $\sum_{j=i}^{n_T} R_{i,j} s_j$ in equation (3.1) before detection. Then, we can obtain the partial Euclidean distance (δ) as follows:

$$\delta_i = \left| \tilde{y}_i - \sum_{j=i}^{n_T} R_{i,j} s_j \right|^2 = \left| \tilde{y}_i - \sum_{j=i}^{n_T} T_{\langle s_j \rangle, \Gamma_{i,j}} \right|^2, \tag{3.2}$$

with $T_{\langle s_j \rangle, \Gamma_{i,j}} = R_{i,j} s_j$. Here, $\langle \cdot \rangle$ gets the integer unique identifier (UID) of a constellation symbol and index $\Gamma_{i,j}$ is the index that occupies the value $R_{i,j}$ in the matrix $\mathbf{T}$. This will allow us to avoid common computation for different possible solutions ($\mathbf{s} \in \mathcal{O}^{n_T}$) decreassing the computational cost of the detection process. This matrix is formed by $M \times n_v$ elements, $T_{i,j}$, which are the result of multiplying the constellation symbol $\mathcal{O}_i$ by the j-th non-zero value of matrix $\mathbf{R}$ (see Fig. 3.10):

$$\mathbf{T} = \begin{bmatrix} \mathcal{O}_1 R_{1,1} & \mathcal{O}_1 R_{1,2} & \dots & \mathcal{O}_1 R_{n_T,n_T} \\ \mathcal{O}_2 R_{1,1} & \mathcal{O}_2 R_{1,2} & \dots & \mathcal{O}_2 R_{n_T,n_T} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{O}_M R_{1,1} & \mathcal{O}_M R_{1,2} & \dots & \mathcal{O}_M R_{n_T,n_T} \end{bmatrix}. \tag{3.3}$$

The number of non-zero values (n_v) of $\mathbf{R}$ can be calculated by a function called nz, which takes as input argument the number of columns of the matrix:

$$n_v = nz(n_T) = \sum_{i=1}^{n_T} i = \frac{n_T(n_T+1)}{2}. \tag{3.4}$$

Then, each row i of the matrix $\mathbf{T}$ contains all non-zero valued elements of matrix $\mathbf{R}$ multiplied by the constellation complex-valued symbol $\mathcal{O}_i$. Fig. 3.10 shows an example for a matrix $\mathbf{R}$ of size 3×3 with $n_v = 6$ non-zero values that are represented as $R_{i,j}^{(t)}$, where t represents the index of its

corresponding column in the $\mathbf{T}$ matrix ($t = \Gamma_{i,j}$). Then, given the i-row and j-column of $\mathbf{R}$ we can obtain the index $\Gamma_{i,j}$ as:

$$\Gamma_{i,j} = ((i-1) \times n_T + i) - nz(i-1) + (j-i). \tag{3.5}$$

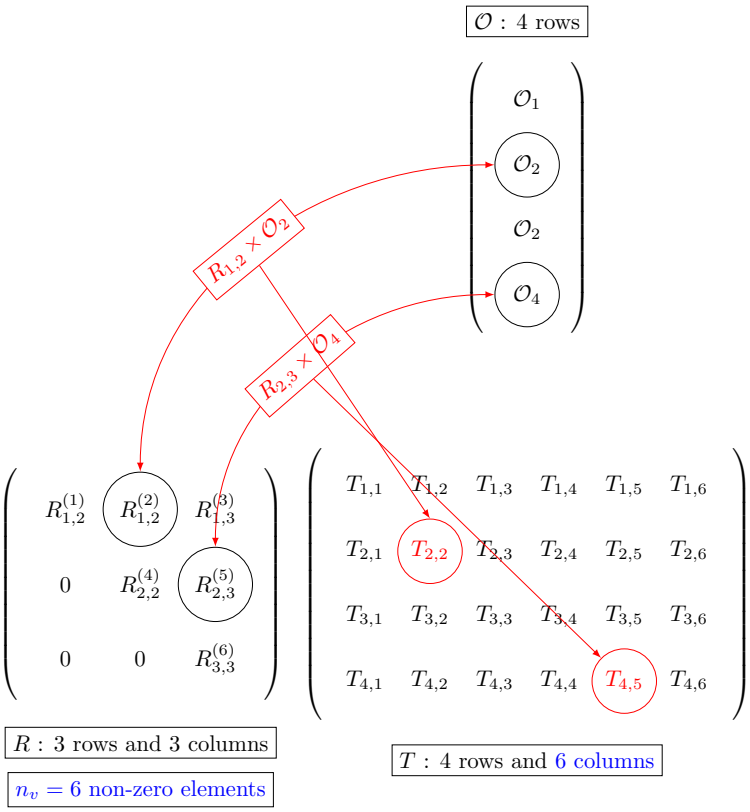


Figure 3.10. Efficient Euclidean Distance Calculation: correspondence between non-zero elements of matrix $\mathbf{R}$ and matrix $\mathbf{T}$ for a 3×3 MIMO system with QPSK constellation.

To quantify the complexity reduction in terms of FLOP (FLoating-point Operations), the problem (3.1) with cost $\Theta(M^{n_T})$ has been considered as target problem. The use of EEDC allows to decrease its asymptotic cost from $\Theta(M^{n_T})$ to $\Theta(M \cdot n_v)$. This performance improvement may be increased if the channel matrix remains constant for several signal transmissions since the $\mathbf{T}$ matrix is calculated only once and reused in the entire

detection process. As an example, let us consider the MLE tree detection shown in Fig. 2.9, where a 2×2 MIMO system using BPSK constellation has been used. The associated triangular matrix has the following structure:

$$\mathbf{R} = \begin{bmatrix} R^1_{1,1} & R^2_{1,2} \\ 0 & R^3_{2,2} \end{bmatrix}.$$

The operations needed to calculate the partial Euclidean distances with and without EEDC are shown in Table 3.1. As can be seen the number of multiplications needed with EEDC is zero whilst if EEDC is not used the number of operation has an as asymptotic cost of $\Theta(M^{n_T})$.

Table 3.1. Number and operations type needed in the MLE detection with and without the use of EEDC in a 2×2 MIMO system with BPSK constellation.

$\mathbf{s} \in \mathcal{O}^{n_T}$	No EEDC	EEDC
$[\mathcal{O}_1, \mathcal{O}_1]$	$\delta_2 = \lvert\tilde{y} - R_{2,2}\mathcal{O}_1\rvert^2$ $\delta_1 = \lvert\tilde{y} - (R_{1,1}\mathcal{O}_1 + R_{1,2}\mathcal{O}_1)\rvert^2$	$\delta_2 = \lvert\tilde{y} - T_{1,3}\rvert^2$ $\delta_1 = \lvert\tilde{y} - (T_{1,1} + T_{1,3})\rvert^2$
$[\mathcal{O}_2, \mathcal{O}_1]$	$\delta_2 = \lvert\tilde{y} - R_{2,2}\mathcal{O}_1\rvert^2$ $\delta_1 = \lvert\tilde{y} - (R_{1,1}\mathcal{O}_2 + R_{1,2}\mathcal{O}_1)\rvert^2$	$\delta_2 = \lvert\tilde{y} - T_{1,3}\rvert^2$ $\delta_1 = \lvert\tilde{y} - (T_{2,1} + T_{1,3})\rvert^2$
$[\mathcal{O}_1, \mathcal{O}_2]$	$\delta_2 = \lvert\tilde{y} - R_{2,2}\mathcal{O}_2\rvert^2$ $\delta_1 = \lvert\tilde{y} - (R_{1,1}\mathcal{O}_1 + R_{1,2}\mathcal{O}_2)\rvert^2$	$\delta_2 = \lvert\tilde{y} - T_{2,3}\rvert^2$ $\delta_1 = \lvert\tilde{y} - (T_{1,1} + T_{2,3})\rvert^2$
$[\mathcal{O}_2, \mathcal{O}_2]$	$\delta_2 = \lvert\tilde{y} - R_{2,2}\mathcal{O}_2\rvert^2$ $\delta_1 = \lvert\tilde{y} - (R_{1,1}\mathcal{O}_2 + R_{1,2}\mathcal{O}_2)\rvert^2$	$\delta_2 = \lvert\tilde{y} - T_{2,3}\rvert^2$ $\delta_1 = \lvert\tilde{y} - (T_{2,1} + T_{2,3})\rvert^2$
Sums:	$M^{n_T} \cdot (1+2) = 12$	$M^{n_T} \cdot (1+2) = 12$
Powers:	$M^{n_T} \cdot (1+1) = 8$	$M^{n_T} \cdot (1+1) = 8$
Multiplations:	$M^{n_T} \cdot (1+2) = 12$	0

Implementation of Hard-Output MIMO Detectors

4

Implementation of Hard-Output MIMO Detectors

4

In MIMO system simulations, the detector can become a bottleneck requiring a significant effort due to the large amount of signals to be received or the high complexity of certain ML detectors. There are several methods in the state of the art devised to recover the information in a MIMO system, but only a selection of these has been implemented until now in MIMOPack. This selection is formed by detectors with mixed complexities and performances intended to deal with multiple use cases with different scenarios and channel conditions. This chapter presents efficient implementations of several hard-output detection schemes, which considerably decrease the computational time required for the data detection stage in MIMO systems. Throughout the chapter some preprocessing algorithms used to reduce the complexity of the detector will be presented. These implementations have been developed with tools and optimization techniques explained in chapter 3. The implementations are evaluated for different system configurations: constellation type, number of transmitted signals and number of transmitting and receiving antennas. And also for different platform configurations (e.g. number of OpenMP threads). Moreover, the computation times of the proposed OpenMP/CUDA implementations are compared with the execution time of the unoptimized version (U) (i.e.

sequential algorithms without Efficient Euclidean Distance Calculation).

4.1 Introduction

As it was already presented in previous chapters, there are some techniques aimed to reduce the computational cost of the detection algorithms:

- QR decomposition: The reduction of the channel matrix to a canonical ($\mathbf{H} = \mathbf{QR}$) form allows to exploit the triangular structure of matrix $\mathbf{R}$ and enables the representation of the detection problem as a Tree-Search-Based problem.
- Efficient Euclidean Distances Calculation (EEDC): it consists in accelerating the computation of the Euclidean Distances building a matrix with all possible values resulting from the multiplication of each constellation symbol by each channel link.
- Preprocessing: In some cases the use of preprocessing techniques before data detection such as column based ordering, can help to improve the performance in terms of bit-error-rate (BER) of suboptimal detectors and also allows to decrease the computational cost of optimal ones.
- High Performance Computing: The new high performance computer architectures (e.g multi-core and GPUs) allow to accelerate applications that require high computational resources.

Obviously, not all of these techniques are mandatory or required when performing detection. Therefore the library enables the selection of different optimization techniques to be used in the detection module. Flow diagram 4.1 shows the necessary steps for the Hard-Output detection. First of all, the columns of the channel matrix are reordered if required (jump 1). After that, the QR decomposition of channel matrix is performed. Once the matrix $\mathbf{R}$ is computed, matrix $\mathbf{T}$ [see Chapter 3, Section 3.4] must be built if the efficient Euclidean Distances calculation (EEDC) is required (jump 2). Finally, the detector is launched and returns the estimated symbols ($\hat{\mathbf{s}}$) for each transmitted signal and its Euclidean distance ($\boldsymbol{\eta}$). To keep the system model unaltered, the detected symbol vectors should be reordered

as $\hat{\mathbf{s}} = \mathbf{P}\hat{\mathbf{s}}$ if reordered detection has been employed (jump 3). Estimates of the transmitted bits ($\hat{\mathbf{b}}$) can be obtained by demapping the estimated symbol vector $\hat{\mathbf{s}}$ to its corresponding bit-labels according to

$$\hat{b}_{i,k} = [\hat{s}_i]_k \quad i = 1, \ldots, n_T, \;\; k = 1, \ldots, m, \tag{4.1}$$

where $[\hat{s}_i]$ denotes the binary-valued representation of the constellation symbol $\hat{s}_i$.

The detector function (assuming no EEDC) has the following call: hard_detector(complex *$\mathbf{R}$, complex *$\tilde{\mathbf{y}}$, mmp_detector *det*, mmp_config *conf*), where mmp_detector and mmp_config structure types contain parameters related to the detector and parallelization, respectively, choosed by the user.

```
typedef struct{
    integer n_cpus; // number of OpenMP threads
    integer n_gpus; // number of GPUs
    double p_w;     // GPUs workload percentage
}mmp_config;
```

```
typedef struct{
    char *name;          // detector name
    char *ordering;     // channel matrix ordering
    integer n_E;    // number of fully expansion levels
    integer K;           // number of survivors
    integer N_iter; // Selected paths in the SFSD-SOE stage
    double *r;          // Initial radius
    bool eedc;              // Indicates if EEDC must be used
}mmp_detector;
```

As we will see throughout the present chapter, the detection problem can be represented as a decision tree such as shown in Fig. 2.9. For ease of explanation and implementation, MIMOPack provides a set of functions for working on the detection tree. A structure, called mmp_path and denoted by $\mathbb{P}$, is used to contain the information of a tree-branch:

```
typedef struct{
    complex *s;
    integer l;
    double η;
}mmp_path;
```

here, **s** contains the tree-path formed by up to n_T constellation symbols and η the accumulated Euclidean Distance up to level l.

Figure 4.2 shows the pseudocode for the detector in sequential, multi-core, multi-GPU and heterogeneus modes [see Chapter 3, Section 3.3] that is explained in the following items:

- Sequential and multi-core mode: steps 5-6 show the sequential or multi-core execution by setting variable n_{cpus} to the desired number of OpenMP threads by the user ($n_{cpus} = 1$ for sequential mode). Function wrapper "*dt_c_name_wrapper*" returns a list ($\mathcal{L}$) of paths with the N_c estimated signal vectors.

- Multi-GPU mode: steps 8-18, allow to execute the detector over multiple GPUs. The number of signals to be computed for each GPU (N_{dev}) are calculated in function of the workload percentage of the GPUs (p_w) and the number of devices selected N_{gpus}. Function wrapper "*dt_cu_name_wrapper*" returns the list of paths with the N_{dev} estimated signal vectors.

- Heterogeneous mode: steps 20-25 make possible the execution in heterogeneus mode. The master thread splits the computation according to the number of GPUs (n_{gpus}) and OpenMP threads (n_{cpus}) selected. One of the threads will distribute the computation of $N_c^{gpus} = N_c \cdot \frac{p_w}{100}$ among the n_{gpus} NVIDIA devices. The other thread is in charge of controlling and creating as many threads as OpenMP threads have been selected and distributing among them the remainder signals ($N_c - N_c^{gpus}$). To carry out this distribution, nested parallelism must be previously activated (step 20).

The main part of the chapter contains implementations considering the simulation of a large number of signals. As we will see later, this situation

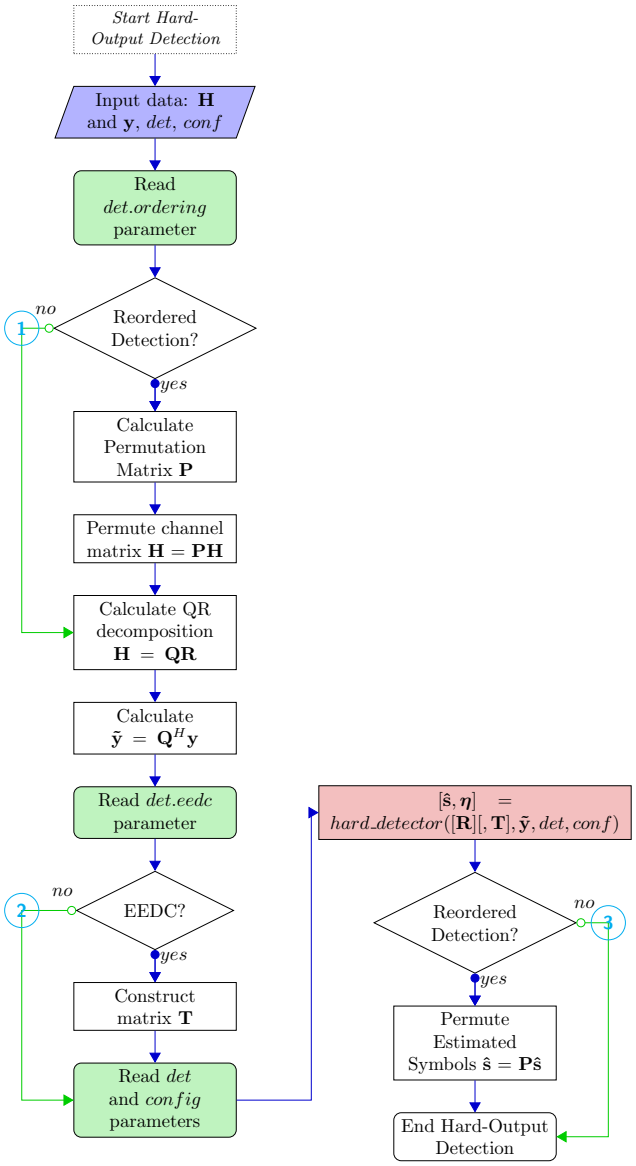


Figure 4.1. Flow Chart of Hard-Output detection.

facilitates the parallelization, since these signals can be independently pro-

cessed. However, we want to provide parallel algorithms that can be used in a real transmission where a single signal vector is processed as a real-MIMO receiver does. For this second use mode, in the last part of the chapter a framework called WinTrees, based on the Divide-and-Conquer paradigm, has been presented. This tool allows and facilitates the free-design and the parallel implementation of tree-search-based MIMO detectors.

The chapter is organized as follows. First, the basic and simplest detector *Zero Forcing with SIC* (ZFSIC) is introduced; this will be used as a preprocessing unit to obtain an initial radius for some of the remaining detectors. Next, the following ML detectors implementations are described: *ML Exhaustive* (MLE), *Schnorr Euchner Sphere Decoder* (SESD) and *Automatic Sphere Decoder* (ASD). Finally, two fixed complexity Tree-Search based detectors are detailed: *K-BEST Sphere Decoder* (K-Best) and *Hard Fixed Complexity Sphere Decoder* (HFSD). The Divide-and-Conquer framework is described at the last part of the chapter. Each description contains the sequential pseudocode, some OpenMP implementation details and the CUDA algorithm version. In order to make an assessment of the MIMOPack Hard-Output detectors, a performance subsection is presented for each detector, which shows time measurements with different configurations taken on the Computer System codename A [see Chapter 3, Section 3.1.3].

4.1.1 OpenMP implementation details

A potential MIMOPack user will use normally the library to carry out Montecarlo simulations where a large amount of bits is considered in order to reproduce those obtained in a real transmission and get average performances. This way of library using mode is highly desired because it allows direct parallelization distributing among the available computational resources the set of signal vectors (N_c) to process (see Fig. 4.3). To simplify the presentation of the algorithms, we have assumed that the channel will remain constant for the entire transmission (i.e $L_{ch} = N_c$). However, the library allows to remove this restriction providing a number of signals N_c multiple of L_{ch}.

Along the chapter, the parts of the algorithms susceptible to be parallelized (i.e. for loops iterators) are accompanied with specific OpenMP pragmas coloured in red. Based on the number of iterations of the possible parallelizable loops in each detector, the library decides the most appropri-

MIMOPack Hard detector pseudocode

Input: $\mathbf{R} \in \mathbb{C}^{n_T \times n_T}$, $\tilde{\mathbf{y}} \in \mathbb{C}^{n_R \times N_c}$, $det \in mmp\_detector$, $conf \in mmp\_config$
Output: $\hat{\mathbf{s}} \in \mathbb{C}^{n_T \times N_c}, \boldsymbol{\eta} \in \mathbb{R}^{N_c}$

function $[\hat{\mathbf{s}}, \boldsymbol{\eta}]$ = hard_detector($\mathbf{R}$, $\tilde{\mathbf{y}}$, det, $conf$)
1. Initialize $\mathcal{L}$ with N_c empty paths
2. $N_c = size(\tilde{\mathbf{y}}, 2)$
3. $N_c^{(gpus)} = N_c * (\frac{conf.p_w}{100})$
4. **if** $N_c^{(gpus)} = 0$
5. **omp_set_num_threads($conf.n_{cpus}$)**
6. $[\mathcal{L}_{N_c^{(gpus)}+1:N_c}] = dt\_c\_name\_wrapper(\mathbf{R}, \tilde{\mathbf{y}}_{:,N_c^{(gpus)}+1:N_c}, det)$
7. **else if** $N_c^{(gpus)} = N_c$
8. **#pragma omp parallel for num_threads($conf.n_{gpus}$)**
9. **for** $dev = 1 : conf.n_{gpus}$
10. $N_{dev} = \frac{N_c^{(gpus)}}{conf.n_{gpus}}$
11. $ini = (dev - 1) \cdot N_{dev} + 1$
12. $end = dev \cdot N_{dev}$
13. **if** $dev = conf.n_{gpus}$
14. $end = N_{dev}$
15. **end**
16. $setDevice(dev)$
17. $[\mathcal{L}_{ini:end}] = dt\_cu\_name\_wrapper(\mathbf{R}, \tilde{\mathbf{y}}_{:,ini:end}, det)$
18. **end**
19. **else**
20. **omp_set_nested(1)**
21. **#pragma omp parallel sections num_threads(2)**
22. **#pragma omp section**
23. Get $[\mathcal{L}_{1:N_c^{(gpus)}}]$ such as steps 8-18
24. **#pragma omp section**
25. Get $[\mathcal{L}_{N_c^{(gpus)}+1:N_c}]$ such as steps 5-6
26. **end**
27. **for** $n = 1 : N_c$
28. $\eta_n = \mathcal{L}_n.\eta$
29. $\hat{\mathbf{s}}_{:,n} = \mathcal{L}_n.\mathbf{s}$
30. **end**
end

Figure 4.2. MIMOPack Hard-Output Detector Pseudocode.

ate loop to parallelize.

Algorithm in Fig. 4.3 is responsible to launch the N_c required detectors which can be processed in parallel keeping the OpenMP pragma in step 2.

C/OpenMP Hard-Output detection wrapper pseudocode

function $[\mathcal{L}]$ = dt_c_name_wrapper($\mathbf{R}$, $\tilde{\mathbf{y}}$, det)
1. $N_c = size(\tilde{\mathbf{y}}, 2)$
2. **#pragma omp parallel for**
3. **for** $n = 1 : N_c$
4. $\quad [path] = dt\_c\_name(\mathbf{R}, \tilde{\mathbf{y}}_{:,n}[, det.r_n][, det.n_E][, det.K])$
5. $\quad \mathcal{L} = \mathcal{L} \cup path$
6. **end**

end

Figure 4.3. C/OpenMP Hard-Output detection wrapper pseudocode. Parameters depend on the chosen detector.

Function *dt_c_name_wrapper* is a generic function used to launch multiple detectors among different signals. For example, if $det.name = mle$, this function takes the name *dt_c_mle_wrapper* and therefore will carry out N_c calls to *dt_c_mle* detector.

Each detector is called with different parameters (inside square brackets) depending on the detector chosen in structure *det*. Each C/OpenMP detector returns a variable, called *path*, which contains the information related to the estimated signal: symbol vector $path.\mathbf{s}$ and its Euclidean distances $path.\eta$. This information is stored for each signal n in $\hat{\mathbf{s}}_{:,n}$ and $\boldsymbol{\eta}_n$.

4.1.2 CUDA implementation details

Before explaining the implementations carried out with CUDA, it seems necessary to explain some implementation details that have been considered and are common for all detectors.

Generic implementations

Nowadays different NVIDIA GPU architectures such as Fermi, Kepler or Maxwell exist in the market. These architectures show increasingly better computation capabilities and new features that usually offer better performance. Throughout the development of this thesis, efficient and highly coupled detectors to a given GPU architecture such as [75][76] have been

developed. However, this type of specific programming requires a hugh effort and many years of development. This is why the algorithms that have been included in the first MIMOPack release are designed to run on any NVIDIA card regardless of the architecture.

Limited GPU Memory Capabilities

Generally all the detectors require to cope with large amounts of data, since it is necessary to store the information of several candidates for all signals. This problem becomes more tricky as the number of signals (N_c) to simulate increases. Although the latest models of NVIDIA cards have 6 GB of Global Memory (which would be enough to store a considerable number of signals), there are other models that only have 1 GB-2 GB.

For this reason in MIMOPack, CUDA kernels are designed to estimate only N_s signals in parallel at once. The parameter N_s will be choosen as a function of the storage capabilities of the graphic card mounted in the target platform. That is, when the memory is enough to store the information required for the detection of N_c signals simultaneously, $N_s = N_c$. Otherwise, when the memory requirements needed to detect them are higher than those of the GPU, N_s will be computed in order to fit into the GPU memory. As it can be seen in Fig. 4.4 this process is carried out using a generic function $gpu\_name\_fitting$ (dependent on the detector name) which takes as inputs: the GPU model, the number of signals to simulate and the MIMO system size. Once N_s has been calculated, the CUDA wrapper makes $\frac{N_c}{N_s}$ calls to the CUDA detector to complete the simulation.

The output of each CUDA detector is a list of N_s paths, called $\mathcal{D}$, which contains the estimated signals calculated by the "dt_cu_name" at once. The list with the total estimated signal $\mathcal{L}$ is updated each time the CUDA detector ends in step 8 in Fig. 4.4.

Moreover, during the detection process, the detector should maintain a list of the symbol vectors that are being estimated, one for each thee-path considered. In order to reduce memory requirements and the cost of data transfers our algorithms keep a list of paths with signal vectors ($path.\mathbf{s}$) in integer format (not complex).

CUDA Hard-Output detection wrapper pseudocode

function $[\mathcal{L}]$ = dt_cu_name_wrapper($\mathbf{R}$, $\tilde{\mathbf{y}}$, det)
1. $[n_T, N_c] = size(\tilde{\mathbf{y}})$
2. $\mathcal{L} = \emptyset$
3. $N_s = gpu\_name\_fitting(model, N_c, n_T)$
4. **for** $n = 1 : \frac{N_c}{N_s}$
5. $\quad ini = (n-1) \cdot N_s + 1$
6. $\quad end = n \cdot N_s$
7. $\quad [\mathcal{D}] = dt\_cu\_name(\mathbf{R}, \tilde{\mathbf{y}}_{:,ini:end}[, det.\mathbf{r}_{ini:end}][, det.n_E][, det.K])$
8. $\quad \mathcal{L} = \mathcal{L} \cup \mathcal{D}$
9. **end**

end

Figure 4.4. CUDA Hard-Output detection wrapper pseudocode.

Grid Configuration

A bidimensional grid configuration with $N_{Bx} = N_B$, $N_{By} = N_B$ blocks per dimension has been considered for all Hard and Soft Detectors (see Fig. 4.5). The number of blocks N_B depends on the number of threads per dimension, which are denoted by N_{tx} and N_{ty}, respectively. The block size will be chosen to be a multiple of 32 in order to avoid incomplete warps. Then the value of N_B is obtained as

$$N_B = \left\lceil \sqrt{\frac{n_{th}}{(N_{tx} \cdot N_{ty})}} \right\rceil . \tag{4.2}$$

As the performance tests have been done on the NVIDIA GPU mounted in the Computer System codename A, the thread per dimension values have been fixed $N_{tx} = N_{ty} = 32$. This value can be modified by the user.

4.1.3 Assessment of parallel algorithms

Some experiments have been carried out in order to evaluate the parallel implementations of the Hard and Soft detectors. This section sumarizes the tools and performance indices used to analyze the efficiency of the parallel algorithms.

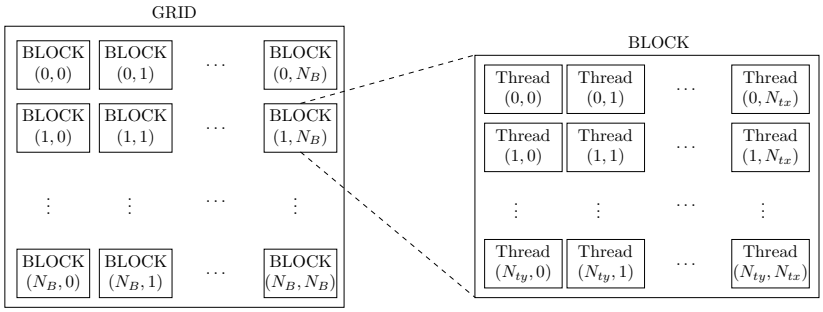


Figure 4.5. Grid distribution considered for the kernels of the Hard and Soft output CUDA detectors.

Computer System for simulation testing

During the development of this thesis, different computer systems have been used for the evaluation of the performance of the developed algorithms. For example, Nvidia Tesla and Fermi GPUs were used for the evaluation of different detectors in [75] and [76] respectively. For the assessment of the library, a machine with multiple Kepler GPUs and multiple multi-core processors has been employed in order to evaluate more platform configurations (i.e multi-core, single-GPU, multi-GPU and multi-core+multi-GPU). The most relevant features and specifications are listed below:

- CPU: Two Intel Xeon CPU E5-2697 at 2.70 GHz
- CPU cores: 12 cores per CPU
- Hyperthreading: Yes
- Operative System: Linux CentOS release 6.5
- Architecture: x86_64
- GPU: Two Tesla K20Xm
- Compute Capability: 3.5
- CUDA SDK: 5.5
- Architecture: Kepler
- Number CUDA of cores: 14 (SM) x 192 cores = 2688 cores

Execution Time

The time execution is a performance index that allows measure the speed of the algorithm. In the case of a sequential program, the execution time (T_S) is the time from the beginning of program execution until it ends. In the case of a parallel program, the execution time (T_P) is the time from the beginning of program execution in the parallel system until the last processor completes its execution.

Speedup

The speedup for p processors, S_P, is the ratio between the time execution of the sequential algorithm, T_S, and time execution version of the parallel algorithm on p processors, T_P. The speedup is computed as:

$$S_P = \frac{T_S}{T_P}. \tag{4.3}$$

This index indicates the speed gain obtained with parallel execution. For example, a speedup equal to 2 indicates that the time is reduced by half running the algorithm with multiple processors.

In the best case, the execution time of a program in parallel with p processors will be p times lower than the execution time on a single processor. Then, the maximum value of S_P of a parallel algorithm is p. Usually, the execution time will never be reduced by the same order p, because extra overhead that appears to solve the problem on multiple processors (e.g. synchronization, dependencies, memory accessess, etc.) have to be considered.

Scalability

Scalability is the ability of an algorithm to maintain its performance proportionately when the number of processors and the size of the problem increases. A scalable parallel algorithm is usually able to maintain constant efficiency when we increase the number of processors even enlarging the problem.

A study of the execution time increasing the number of processors with different number of signals, system and constellations sizes was carried out to evaluate the influence of these parameters on the algorithm performance.

4.2 Zero Forcing SIC Detector Implementation

The *Zero Forcing* (ZF) detector is a linear and simple technique for recovering the transmitted signals at the receiver [29] as was introduced in chapter 2, section 2.3.1. The estimated transmitted vector can be obtained by means of the pseudo-inverse of the channel matrix as follows:

$$\hat{\mathbf{s}} = \mathcal{Q}\left\{\mathbf{H}^{\dagger}\mathbf{y}\right\} = \mathcal{Q}\left\{(\mathbf{H}^{H}\mathbf{H})^{-1}\mathbf{H}^{H}\mathbf{y}\right\}, \tag{4.4}$$

where $(\cdot)^{\dagger}$ stands for the pseudo-inverse matrix, $(\cdot)^{-1}$ indicates simple matrix inversion and the function $\mathcal{Q}(\cdot)$ assigns the closest modulation constellation symbol.

The computation of the inverse matrix has a high computational complexity. The QR decomposition can be used for matrix inversion to reduce the complexity of this method. If $\mathbf{H} = \mathbf{QR} = \mathbf{Q}\begin{bmatrix}\mathbf{R}_1 \\ 0\end{bmatrix}$ with $\mathbf{R}_1 \in \mathbb{C}^{n_T \times n_T}$ is used, $(\mathbf{H}^H\mathbf{H})^{-1}\mathbf{H}^H\mathbf{y} = ((\mathbf{QR})^H(\mathbf{QR}))^{-1}(\mathbf{QR})^H\mathbf{y} = (\mathbf{R}_1^H\mathbf{R}_1)^{-1}[\mathbf{R}_1^H \ \ 0]\tilde{\mathbf{y}} = \mathbf{R}_1^{-1}\tilde{\mathbf{y}}_1$ with $\tilde{\mathbf{y}} = \mathbf{Q}^H\mathbf{y}$, $\tilde{\mathbf{y}} = \begin{bmatrix}\tilde{\mathbf{y}}_1 \\ \tilde{\mathbf{y}}_2\end{bmatrix}$ and $\tilde{\mathbf{y}}_1 \in \mathbb{C}^{n_T \times 1}$. For the sake of simplicity, submatrices $\mathbf{R}_1$ and $\tilde{\mathbf{y}}_1$ are renamed as $\mathbf{R}$ and $\tilde{\mathbf{y}}$ respectively. In this case, the quantization and detection are done for each component of $\hat{\mathbf{s}}$ successively and not jointly as in the ZF as:

$$\hat{s}_i = \mathcal{Q}\left\{\frac{\tilde{y}_i - \sum_{j=i+1}^{n_T} R_{i,j}\hat{s}_j}{R_{i,i}}\right\}, \quad i = n_T, \ldots, 1. \tag{4.5}$$

This technique is known as *Succesive Interference Cancellation* (SIC) or *decision feedback* and not only reduces the computational complexity of the method but also decreases the noise amplification of linear detectors improving the performance of the *Zero Forcing* method [30]. In the decision-tree perspective, ZF-SIC can be seen as just exploring one single path down from the root (see Fig. 4.6). After n_T steps we end up at one of the leaf nodes (coloured in red), but not necessarily in the one with the smallest euclidean distance (coulored in blue).

Figure 4.7 shows the necessary steps to estimate a signal with the ZF-SIC detector. Note that, *Efficient Euclidean Distance Calculation* (EEDC)

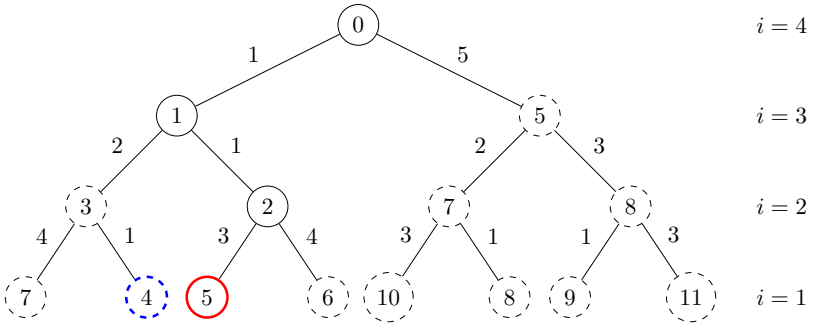


Figure 4.6. Decoding tree of the Zero Forcing with SIC algorithm for a 3×3 MIMO system with BPSK symbols.

ZF-SIC pseudocode

function $[path] =$ dt_c_zfsic($\mathbf{R}$, $\tilde{\mathbf{y}}$)
1. $n_T = size(\mathbf{R}, 1)$
2. **for** $i = n_T : -1 : 1$
3. $\quad path.s_i = \mathcal{Q}\left\{\frac{\tilde{y}_i - \sum_{j=i+1}^{n_T} R_{i,j} path.s_j}{R_{i,i}}\right\}$
4. **end**
5. $path.\eta = \|\tilde{\mathbf{y}} - \mathbf{R} path.\mathbf{s}\|^2$

end

Figure 4.7. Zero Forcing with Successive Interference Cancellation Pseudocode.

can be used in order to decrease the computational cost of the inner sumatory in step 3) making $R_{i,j} path.s_j = T_{\langle path.s_j \rangle, \Gamma_{i,j}}$, where $\Gamma_{i,j}$ represents the index of the column that the $R_{i,j}$ value occupies in matrix $\mathbf{T}$ [see Chapter 3, Section 3.4]. Function $\langle \cdot \rangle$ gets the integer unique identifier (UID) of a constellation symbol. This mapping is straightforwardly done in practice because the symbol vectors used in the detection process are stored in integer format. An example of detection with EEDC is shown below, for a system with $n_T = n_R = 4$ and 16-QAM modulation. The matrix $\mathbf{R}$ (eq. 4.6) after QR decomposition has $n_v = 10$ non-zero values, which are represented as $R_{i,j}^{(t)}$, where $t = \Gamma_{i,j}$.

$$\mathbf{R} = \begin{bmatrix} R_{1,1}^{(1)} & R_{1,2}^{(2)} & R_{1,3}^{(3)} & R_{1,4}^{(4)} \\ 0 & R_{2,2}^{(5)} & R_{2,3}^{(6)} & R_{2,4}^{(7)} \\ 0 & 0 & R_{3,3}^{(8)} & R_{3,4}^{(9)} \\ 0 & 0 & 0 & R_{4,4}^{(10)} \end{bmatrix}. \tag{4.6}$$

The detection process (see Fig. 4.8) starts estimating $\hat{s}_4 = \mathcal{O}_7$. This symbol is used to obtain the next component $\hat{s}_3$ and so on. In this case we can calculate the sumatory adding different elements of $\mathbf{T}$.

Iteration 1. $\hat{s}_4 = \mathcal{Q}\left\{\frac{\tilde{y}_4}{R_{4,4}}\right\} = \mathcal{O}_7$

Iteration 2. $\hat{s}_3 = \mathcal{Q}\left\{\frac{\tilde{y}_3 - R_{3,4}\mathcal{O}_7}{R_{3,3}}\right\} = \mathcal{Q}\left\{\frac{\tilde{y}_3 - T_{7,9}}{R_{3,3}}\right\} = \mathcal{O}_{10}$

Iteration 3. $\hat{s}_2 = \mathcal{Q}\left\{\frac{\tilde{y}_2 - (R_{2,3}\mathcal{O}_{10} + R_{2,4}\mathcal{O}_7)}{R_{2,2}}\right\} = \mathcal{Q}\left\{\frac{\tilde{y}_2 - (T_{10,6} + T_{7,7})}{R_{2,2}}\right\} = \mathcal{O}_3$

Iteration 4. $\hat{s}_1 = \mathcal{Q}\left\{\frac{\tilde{y}_1 - (R_{1,2}\mathcal{O}_3 - R_{1,3}\mathcal{O}_{10} + R_{1,4}\mathcal{O}_7)}{R_{1,1}}\right\} = \mathcal{Q}\left\{\frac{\tilde{y}_1 - (T_{3,2} + T_{10,3} + T_{7,4})}{R_{2,2}}\right\} = \mathcal{O}_1$

Figure 4.8. Successive Interference Cancellation detection process with EEDC: example with a 4×4 MIMO system and 16-QAM constellation.

ZF-SIC detector is difficult to parallelize, because a determined component $\hat{s}_i$ needs the computation of previous components and the size of matrix $\mathbf{R}$ uses to be small. For this reason, the parallelism of this algorithm is based on the estimation of a particular signal $\hat{\mathbf{s}}_{:,n}$ per thread as Fig. 4.3 shows.

4.2.1 CUDA Implementation

As occurs in the OpenMP implementation, the parallelism of this algorithm is based on the estimation of a unique signal per thread. The proposed ZF-SIC CUDA implementation is composed by a single kernel, which must estimate N_s signals in parallel. Algorithm described in Fig. 4.9 shows the required steps to launch the kernel.

First, it is necessary to allocate memory and copy the input ($\mathbf{T}$, $\mathbf{R}$, $\tilde{\mathbf{y}}$) matrices. The output list shall contain the estimated vectors $\mathcal{L}$ related to the N_s signals and must be allocated in the GPU global memory (GM) before starting the kernel execution. Here, $\mathbb{P}$ denotes the data type *mmp_path*

ZF-SIC Kernel-Launcher Pseudocode

Input: $\mathbf{T} \in \mathbb{C}^{M \times n_v}$, $\mathbf{R} \in \mathbb{C}^{n_T \times n_T}$, $\tilde{\mathbf{y}} \in \mathbb{C}^{n_R \times N_s}$
Output: $\mathcal{L} \in \mathbb{P}^{N_s}$

function $[\mathcal{L}]$ = dt_cu_zfsic($\mathbf{R}$, $\tilde{\mathbf{y}}$)
1. Allocate and copy $\mathbf{T}$, $\mathbf{R}$ and $\tilde{\mathbf{y}}$ in GPU GM
2. Allocate output data $\mathcal{L}$ in GPU GM
3. Copy constellation symbols $\mathcal{O}$ in GPU CM
4. Select block and grid configuration with $n_{th} = N_s$
5. $[\mathcal{L}] = kernel\_zfsic(\mathbf{T}, \mathbf{R}, \tilde{\mathbf{y}})$
6. Copy $\mathcal{L}$ in CPU

end

Figure 4.9. Zero Forcing SIC Kernel-launcher for N_s signals. This launcher calls the kernel in Fig. 4.10.

structure with the field **s** in integer format.

As the matrices used in MIMO systems can be considered quite small, the main computational cost source is the step of quantizing to obtain the estimated symbol at each level. This function goes through the constellation to return the symbol. Constellation symbols will not change during the execution and are read only, then they are very suitable to be kept in constant memory (CM). From now on it will be assumed that the input, auxiliary and output variables are stored in GPU global memory. Only variables that can be stored in constant memory will be marked out for the explanation of the algorithms.

Figure 4.10 shows the kernel_zfsic pseudocode, where each CUDA thread applies the ZF-SIC method using the n-th received signal. Then, the number of threads necessary for the appropiatte grid dimension is $n_{th} = N_s$. The output of the kernel is a list which contains the estimated vector $\mathcal{L}_n.\mathbf{s}$ and its Euclidean Distance $\mathcal{L}_n.\eta$ for each processed signal n.

4.2.2 Performance Results

Some experiments have been carried out in order to compare the parallel implementations of the hard-output ZFSIC on multi-core and GPU. Its execution time was compared to the unoptimized and sequential implemen-

KERNEL ZFSIC Pseudocode

function $[\mathcal{L}]$ = kernel_zfsic($\mathbf{T}$, $\mathbf{R}$, $\tilde{\mathbf{y}}$)
1. Calculate using the thread global index z:
2. $\quad \cdot$ Time slot identifier $n \in \{1, ..., N_s\}$
3. **if** $n < N_s$
4. $\quad$ **for** $i = n_T : -1 : 1$
5. $\quad\quad x = \mathcal{Q}\left\{ \frac{\tilde{y}_{i,n} - \sum_{j=i+1}^{n_T} T_{\mathcal{L}_n.s_j, \Gamma_{i,j}}}{R_{i,i}} \right\}$
6. $\quad\quad \mathcal{L}_n.s_i = \langle x \rangle$
7. $\quad$ **end**
8. $\quad \mathcal{L}_n.\eta = \sum_{i=n_T}^{1} \left| \tilde{y}_{i,n} - \sum_{j=i}^{n_T} T_{\mathcal{L}_n.s_j, \Gamma_{i,j}} \right|^2$
9. **end**

end

Figure 4.10. Zero Forcing SIC detection by the z-th thread for N_s signals called from Kernel-Launcher in Fig. 4.9

tation (i.e. without the EEDC optimization) codename U.

Table 4.1 shows the execution time in miliseconds for a 6×6 system with 16-QAM symbols as a function of the simulated signals (N_c). The speedups (S_P) are defined as the ratio between the execution time of unoptimized version (U) and the rest of versions presented in this section. Version (U) GPU refers to the parallel GPU version without EEDC.

It can be observed that, generally, the higher the number of signals, the higher the achieved speedup for the OpenMP versions. However, the performance degrades drastically using 32 and 48 threads caused for the hyperthreading technology (HT). HT creates for each physical processor two virtual or logical cores, and shares the workload between them when possible. The execution time is increased due to the additional costs of creation, management and memory accesses of these virtual threads. Note that, the gain obtained using EEDC in this detector improves up to 10% the detector performance (see execution time for 1 OpenMP threads in Table 4.1).

The experimental GPU measurements in Table 4.1 show as CUDA version fails to accelerate the sequential version when the number of signals is small. This is due to the lower complexity and the non parallel pattern of the ZFSIC detector. This problem gradually disappears when the

complexity of the detection stage increases, for example with the number of tranmistter antennas (n_T). In this escenario, where very large MIMO arrays are considered, the CUDA detector is up to 7 times faster than unoptimized version (see Fig. 4.11).

Table 4.1. Time Execution comparison in miliseconds and speedup (S_P) of ZFSIC detector with different library configurations for a 6×6 system using 16-QAM constellation as a function of the simulated signals (N_c).

		$N_c = 1000$		$N_c = 10000$		$N_c = 100000$	
		Time[ms]	S_P	Time[ms]	S_P	Time[ms]	S_P
	U	0.82	1.00	7.96	1.00	71.24	1.00
OpenMP Threads	1	0.75	1.09	7.51	1.06	65.03	1.10
	2	0.69	1.19	5.92	1.34	37.71	1.89
	4	0.65	1.26	4.94	1.61	20.54	3.47
	8	1.40	0.59	1.40	2.35	5.69	6.67
	12	1.38	0.59	1.42	2.18	5.61	9.50
	24	3.18	0.26	1.78	1.71	4.47	10.70
	32	3.84	0.21	2.90	1.30	2.74	4.64
	48	30.35	0.03	46.00	0.30	0.17	1.86
	GPU	208.95	$0.39 \cdot 10^{-2}$	235.27	0.03	23.19	3.07
	(U) GPU	200.80	$0.40 \cdot 10^{-2}$	203.74	0.04	23.06	3.09

When the EEDC is used in GPU version, the time execution is not reduced but also increases (see execution times for GPU and (U) GPU versions in Table 4.1). This is mainly due to the time required to transfer the matrix $\mathbf{T}$ to the GPU memory since can be higher than the gain obtained from its use. Nevertheless, when the system size increases the transfer time and the reduction achieved with EEDC are equated (see Fig. 4.11). In any case, the use of the EEDC optimization in CUDA seems not adequate for the ZFSIC detector.

4.3 ML Exhaustive Detector Implementation

Let us consider the problem

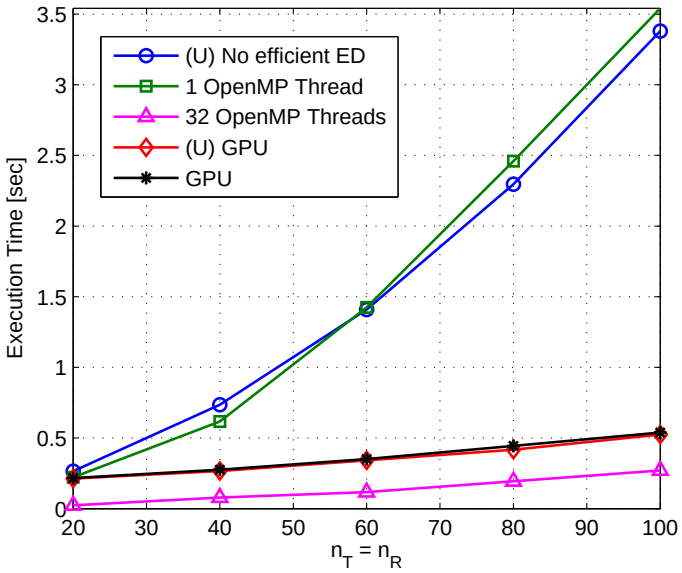


Figure 4.11. Time Execution comparison in seconds of the unoptimized ZFSIC detector to the fastest OpenMP and GPU implementation for a $n_R \times n_T$ system with 16-QAM constellation and $N_c = 100000$.

$$\hat{\mathbf{s}} = \arg \min_{\mathbf{s} \in \mathcal{O}^{n_T}} \|\mathbf{y} - \mathbf{H}\mathbf{s}\|^2, \tag{4.7}$$

where, $\mathcal{O}$ denotes the constellation of size $|\mathcal{O}| = 2^m = M$. The *Maximum-Likehood Exhaustive* (MLE) detector can directly solve problem (4.7) by an exhaustive search over the set of n_T-dimensional lattice points $\mathbf{s} \in \mathcal{O}^{n_T}$. Using the QR decomposition such as in section 2.3.1, problem (4.7) can be expressed as:

$$\hat{\mathbf{s}} = \arg \min_{\mathbf{s} \in \mathcal{O}^{n_T}} \left\{ \sum_{i=n_T}^{1} \left| \tilde{y}_i - \sum_{j=i}^{n_T} R_{i,j} s_j \right|^2 \right\}, \tag{4.8}$$

or alternatively if EEDC is considered:

$$\hat{\mathbf{s}} = \arg \min_{\mathbf{s} \in \mathcal{O}^{n_T}} \left\{ \sum_{i=n_T}^{1} \left| \tilde{y}_i - \sum_{j=i}^{n_T} T_{\langle s_j \rangle, \Gamma i, j} \right|^2 \right\}. \tag{4.9}$$

This problem can be represented as a decision tree, such as figure 4.12, with n_T+1 layers and M branches emerging from each non-leaf node. Each path can be identified with a unique identifier denoted by $q \in \{1, \ldots, M^{nT}\}$. The complete q-th tree-path can be easly obtained by using function as $\mathbf{s} = tree\_path(q, \mathcal{O}, n_T)$ described in Figure 4.13.

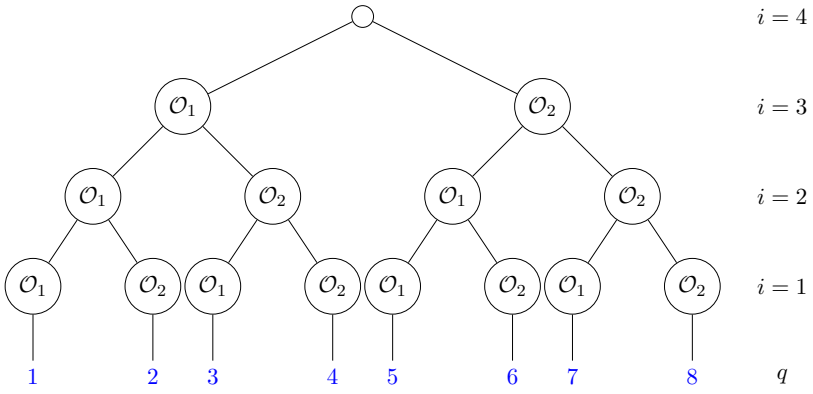


Figure 4.12. Decoding tree of the MLE algorithm for a 3×3 MIMO system with BPSK symbols.

q-th Tree Path

function [**s**] = tree_path(q, $\mathcal{O}$, n)
1. $M = length(\mathcal{O})$
2. $q = q - 1$
3. **for** $i = 1, 2, ..., n$
4. $\quad s_i = \mathcal{O}_{rem(q,M)+1}$,
5. $\quad q = \lceil \frac{q}{M} \rceil$
6. **end**

end

q-th Tree Path in integer format

function [**s**] = tree_path(q, n)
1. $M = length(\mathcal{O})$
2. $q = q - 1$
3. **for** $i = 1, 2, ..., n$
4. $\quad s_i = rem(q, M) + 1$,
5. $\quad q = \lceil \frac{q}{M} \rceil$
6. **end**

end

Figure 4.13. Pseudocode of *tree_path* function: gets n consecutive constellation symbols of the q-th tree-path.

MLE pseudocode

function $[path] = \text{dt\_c\_mle}(\mathbf{R}, \tilde{\mathbf{y}})$
1. $\mathcal{L} = \emptyset$
2. **#pragma omp parallel for**
3. **for** $q = 1 : M^{n_T}$
4. $\quad path.\mathbf{s} = tree\_path(q, \mathcal{O}, n_T)$
5. $\quad path.\eta = \|\tilde{\mathbf{y}} - \mathbf{R}path.\mathbf{s}\|^2$
6. $\quad \mathcal{L} = \mathcal{L} \cup path$
7. **end**
8. $path = min(\mathcal{L}, 1)$

end

Figure 4.14. ML Exhaustive Pseudocode.

Fig. 4.14 illustrates the necessary steps to estimate a signal vector with the MLE detector. The main loop performs an exhaustive search over the total possible symbol vectors $path.\mathbf{s} \in \mathcal{O}^{n_T}$. Once the constellation symbols for a given q-th tree-path have been obtained, its Euclidean Distance is calculated in step 5. If EEDC is being considered, η is calculated adding different elements of the pre-built matrix $\mathbf{T}$ [see Chapter 3, Section 3.4]. The algorithm maintain a list $\mathcal{L}$ with the paths that have been considered up to now. This list is updated every time a new path is explored in step 6. Finally, we can obtain the ML solution searching the path with the minimim ED. This is done in step 8 by using function $min(\mathcal{L}, n_K)$, which returns the n_K minimum paths of $\mathcal{L}$. In the MLE detector, this function is used with $n_K = 1$ in order to find the lattice point that minimizes (4.7).

The calculation of all combinations in the main loop can be executed in parallel. Then, the M^{n_T} iterations can be distributed among the threads by using the pragma *omp parallel for* as a simple way to parallelize the algorithm. Note that in cases where M and n_T are too small, the algorithm performance can decrease specially when it uses a significant number of threads because the workload cannot exploit all the computational resources. To overcome this problem, the library decides the most appropriate loop to parallelize when the number of signals to simulate N_c is greater than M^{n_T}. Thus the parallelized loop is the one that runs the buffer with the received signals during N_c instants (see Fig. 4.3). On the other hand when

we have a suitable number of possible combinations, the MLE detector will be parallelized keeping the pragma in step 2.

4.3.1 CUDA Implementation

The proposed MLE GPU implementation is composed of one single kernel that parallelizes the steps 3-7 of Fig. 4.14 estimating $N_s \times M^{n_T}$ paths at once. In Kernel 4.16 each thread computes the accumulated Euclidean distance $\mathcal{L}_{n,q}.\eta$ for a given signal n and the q-th path considered. Therefore $n_{th} = N_s \cdot M^{n_T}$ threads are needed to explore every tree-branch. Once the kernel has finished the computation, the CPU gets the list with the candidate paths and obtains the minimum path for each signal vector (see step 7 in Fig.4.15).

MLE Kernel-Launcher Pseudocode

Input: $\mathbf{T} \in \mathbb{C}^{M \times n_v}$, $\tilde{\mathbf{y}} \in \mathbb{C}^{n_R \times N_s}$
Output: $\mathcal{F} \in \mathbb{P}^{N_s}$

function $[\mathcal{F}]$ = dt_cu_mle($\mathbf{T}$, $\tilde{\mathbf{y}}$)
1. Allocate and copy $\mathbf{T}$ and $\tilde{\mathbf{y}}$ in GPU GM
2. Allocate output data $\mathcal{L} \in \mathbb{P}^{N_s \times M^{n_T}}$ in GPU GM
3. Select block and grid configuration with $n_{th} = N_s \cdot M^{n_T}$
4. $[\mathcal{L}]$ = kernel_mle($\mathbf{T}$, $\tilde{\mathbf{y}}$)
5. Copy $\mathcal{L}$ in CPU
6. **for** $n = 1 : N_s$
7. $\quad \mathcal{F}_i = min(\mathcal{L}_{i,:}, 1)$
8. **end**

end

Figure 4.15. ML Exhaustive Kernel-launcher for N_s signals. This launcher calls the kernel function in Fig. 4.16.

4.3.2 Performance Results

The performance of the MLE detector has been investigated for a 6×6 system using 16-QAM constellation and considering $N_c = 1000$ signals transmitted. Analyzing the experimental results depicted in Table 4.2, we

Kernel MLE Pseudocode

function $[\mathcal{L}]$ = kernel_mle($\mathbf{T}$, $\tilde{\mathbf{y}}$)
1. Calculate using the thread global index z:
2. · Time slot identifier $n \in \{1, ..., N_s\}$
3. · Tree-Path identifier $q \in \{1, ..., M^{n_T}\}$
4. **if** $n < N_s$
5. $\mathcal{L}_{n,q}.\mathbf{s} = tree\_path(q, n_T)$,
6. $\mathcal{L}_{n,q}.\eta = \sum_{i=n_T}^{1} \left| \tilde{y}_{i,n} - \sum_{j=i}^{n_T} T_{\mathcal{L}_{n,q}.\mathbf{s}_j, \Gamma_{i,j}} \right|^2$
7. **end**

end

Figure 4.16. Calculation of one of the branches of the MLE detector by the z-th thread for N_s signals. This kernel is called from Kernel-Launcher in Fig. 4.15.

can find that the EEDC optimization improves the performance up to 26% for the OpenMP version and $\frac{T_{\text{(U) GPU}}}{T_{\text{GPU}}} = \frac{1.28 \cdot 10^5}{1.11 \cdot 10^5} \approx 15\%$ for the GPU one. If we compare with the results obtained in the ZFSIC detector which achieved a improvement of the 10%, we can see how the MLE detector takes more benefit from this optimization. This is due to two main reasons. On the one hand, the number of PED calculated for MLE is much higher than those calculated by the ZFSIC. On the other hand, the ZFSIC needs both matrices ($\mathbf{T}$ and $\mathbf{R}$) to quantize the estimated symbol. Thus, causing more conflicts in the cache memory and increasing the data transfer costs in the GPU version.

It can be appreciated the good scalability of the MLE detector, since the speedup increases with the number of OpenMP threads. In this case the hyperthreading should be enabled because improves the performance (up to 27 times faster than sequential execution). GPU implementation using EEDC boost the performance of the multi-core version achieving a speedup of 29. Note that, the execution time of unoptimized version (U) is dramatically reduced from 54 minutes to approximately 100 seconds.

Table 4.2. Time Execution in miliseconds and speedup (S_P) of the MLE detectors for a 6×6 system using 16-QAM constellation with $N_c = 1000$.

		Time[ms]	S_P
	U	$32.56 \cdot 10^5$	1.00
OpenMP Threads	1	$25.71 \cdot 10^5$	1.26
	2	$13.29 \cdot 10^5$	2.44
	4	$6.75 \cdot 10^5$	4.82
	8	$3.67 \cdot 10^5$	8.87
	12	$2.49 \cdot 10^5$	13.07
	24	$1.68 \cdot 10^5$	19.38
	32	$1.49 \cdot 10^5$	21.85
	48	$1.21 \cdot 10^5$	26.90
	GPU	$1.11 \cdot 10^5$	29.33
	(U) GPU	$1.28 \cdot 10^5$	25.43

4.4 Schnorr-Euchner SD Implementation

The *Schnorr-Euchner Sphere Decoder* (SESD) performs a search from the top to the bottom of the tree with radius reduction. The sphere decoding methods arise aiming to reduce the range of search, seeking uniquely those lattice points that lie in a hypershera of radius r, around the received vector $\mathbf{y}$. The SD search can be expressed introducing this constraint in (4.8) as follows:

$$\hat{\mathbf{s}} = \arg \min_{\mathbf{s} \in \mathcal{O}^{n_T}} \left\{ \sum_{i=n_T}^{1} \left| \tilde{y}_i - \sum_{j=i}^{n_T} R_{i,j} s_j \right|^2 \leq r \right\}, \tag{4.10}$$

or if EEDC is considered:

$$\hat{\mathbf{s}} = \arg \min_{\mathbf{s} \in \mathcal{O}^{n_T}} \left\{ \sum_{i=n_T}^{1} \left| \tilde{y}_i - \sum_{j=i}^{n_T} T_{\langle s_j \rangle, \Gamma_{i,j}} \right|^2 \leq r \right\}. \tag{4.11}$$

Figure 4.17 shows the pseudocode of the SESD algorithm. As can be seen, it is similar to the MLE detector. However the detector works in this

case with a initial radius, which is used to discard solutions with higher Euclidean Distances. The algorithm uses a path, called *minpath*, to store the path with the minimum ED explored so far, thus at the end *minpath* contains the lattice point that minimizes Eq. (4.10). This path takes as input an initial radius r and is updated each time a leaf node (*path*) is reached with cumulative metric smaller than $minpath.\eta$. The *minpath* is updated setting $minpath.\eta = path.\eta$, and continues exploring the tree with a smaller sphere radius. Thus the algorithm will not consider branches from nodes with ED larger than $minpath.\eta$ (see step 6). In this way, invalid points are discarted since after having explored a certain point in the search set, the algorithm must be only interested in visiting those points that are even closer to the target than the recently visited points.

SESD pseudocode

function [*path*] = dt_c_sesd($\mathbf{R}$, $\tilde{\mathbf{y}}$, r)

1. $minpath.\eta = r$
2. **for** $q = 1 : prune : M^{n_T}$
3. $path.\mathbf{s} = tree\_path(q, \mathcal{O}, n_T)$
4. **for** $i = n_T : -1 : 1$
5. $path.\eta = path.\eta + \left|\tilde{y}_i - \sum_{j=i}^{n_T} R_{i,j} path.s_j\right|^2$
6. **if** $path.\eta > minpath.\eta$
7. break
8. **end**
9. **end**
10. **if** $i = 0$ *//Leaf node visited*
11. $minpath.\mathbf{s} = path.\mathbf{s}$
12. $minpath.\eta = path.\eta$
13. **end**
14. $prune = M^{i-1}$
15. **end**
16. $path = minpath$

end

Figure 4.17. Schnorr-Euchner Sphere Decoder Pseudocode.

When the Accumulated Euclidean Distance for a particular tree-branch, denoted by q, in a particular tree level i is greater than the current radius $minpath.\eta$, the algorithm can reject nodes and subtrees with higher distances in the next search thus accelerating the detection process. The number of consecutive branches to be descarted can be calculated as $prune = M^{i-1}$.

Figure 4.18 shows an example of detection for a 4×4 MIMO system with BPSK constellation ($M = 2$). As can be seen the accumulated Euclidean distance of branch $q = 3$ in level $i = 2$ is 9, which is larger than the actual radius 8 (framed in red). Then, the next branch to be computed is $q = 3 + M^{i-1} = 5$. In the same way for branch $q = 5$ at level $i = 3$ the tree-paths to be discarded (rectangular nodes) can be calculated as $M^{i-1} = 2^2 = 4$. In order to accelerate the computation of the next node to be visited, a vector called $\mathcal{P}$ is created before the detection as

$$\mathcal{P}_i = M^{i-1} \quad \text{for } i = 1, \ldots, n_T + 1 \tag{4.12}$$

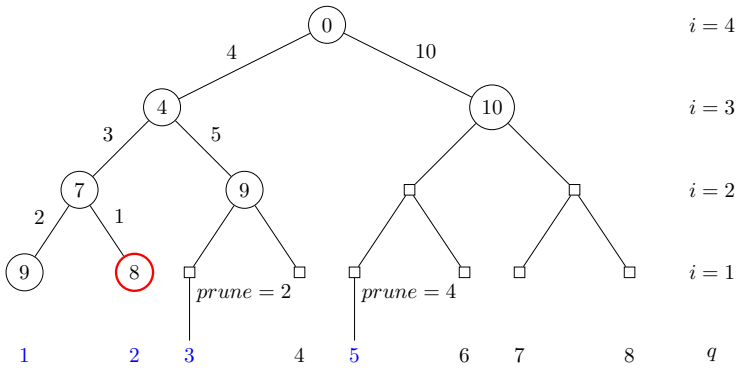


Figure 4.18. Decoding tree of the SESD algorithm for a 3×3 MIMO system with BPSK symbols.

In contrast to the MLE detector, the main loop can not be parallelized since the SESD updates adaptively the search radius when a new leaf node is reached. Then, OpenMP implementation distributes the computation of the received signals in N_c time instants among the OpenMP threads (see Fig. 4.3.

4.4.1 CUDA Implementation

The proposed SESD GPU implementation is composed of one single kernel, which is responsible for the estimation of N_s signals in parallel. Figure 4.19 shows the steps needed by the algorithm to launch the kernel. First, it is necessary to allocate and copy $\mathbf{T}$ and $\tilde{\mathbf{y}}$ matrices. Output ($\mathcal{L}$) and auxiliary ($\mathcal{D}$) lists related to the N_s signals must also be previously allocated. The pruning pattern $\mathcal{P}$ will not change during the execution and is read only, therefore it can be stored in constant memory (GPU-CM). Next, the Kernel 4.20 is launched with the appropiate grid dimension.

SDSE Kernel-Launcher Pseudocode

Input: $\mathbf{T} \in \mathbb{C}^{M \times n_v}$, $\tilde{\mathbf{y}} \in \mathbb{C}^{n_R \times N_S}$, $\mathcal{L} \in \mathbb{P}^{N_s}$
Output: $\mathcal{L} \in \mathbb{P}^{N_s}$

function $[\mathcal{L}]$ = dt_cu_sesd($\mathbf{T}$, $\tilde{\mathbf{y}}$, $\mathcal{L}$)
1. Allocate and copy $\mathbf{T}$ and $\tilde{\mathbf{y}}$ in GPU GM
2. Allocate and copy input/output data $\mathcal{L}$ in GPU GM
3. Allocate auxiliary data $\mathcal{D} \in \mathbb{P}^{N_S}$ in GPU GM
4. Copy prunning pattern $\mathcal{P}$ in GPU CM
5. Select block and grid configuration with $n_{th} = N_S$
6. $[\mathcal{L}]$ = kernel_sesd($\mathbf{T}$, $\tilde{\mathbf{y}}$, $\mathcal{L}$, $\mathcal{D}$)
7. Copy $\mathcal{L}$ in CPU

end

Figure 4.19. Schnorr-Euchner Sphere Decoder Kernel-launcher for N_s signals. This launcher calls the kernel in Fig. 4.20.

As mentioned in the OpenMP implementation, the inner loop can not be parallelized since the radius changes when a new leaf node is reached, for this reason we can only parallelize the first loop. Each CUDA thread explores the tree (for each signal n) to find the ML solution considering all possible alternatives and discarding those showing a distance larger than the current radius. Note that, list of paths $\mathcal{L}$ acts as input and output of the kernel, since contains at the input the initial radius for each signal. For this reason should be copied in GPU memory before the launch of kernel

(see step 2). This list is also used to store the ML solution for each signal vector processed. A list of paths, called $\mathcal{D}$, is used as an auxiliary list where the information of each branch considered during the tree exploration will be stored.

KERNEL SDSE Pseudocode

function $[\mathcal{L}]$ = kernel_sesd($\mathbf{T}$, $\tilde{\mathbf{y}}$, $\mathcal{L}$, $\mathcal{D}$)
1. Calculate using the thread global index z:
2. · Time slot identifier $n \in \{1, ..., N_s\}$
3. **if** $n < N_s$
4. **for** $q = 1 : prune : M^{n_T}$
5. $\mathcal{D}_n.\mathbf{s} = tree\_path(q, n_T)$
6. **for** $i = n_T : -1 : 1$
7. $\mathcal{D}_n.\eta = \mathcal{D}_n.\eta + \left|\tilde{y}_{i,n} - \sum_{j=i}^{n_T} T_{\mathcal{D}_n.\mathbf{s}_j,\Gamma_{i,j}}\right|^2$
8. **if** $\mathcal{D}_n.\eta > \mathcal{L}_n.\eta$
9. break
10. **end**
11. **end**
12. **if** $i = 0$ *//Leaf node visited*
13. $\mathcal{L}_n.\mathbf{s} = \mathcal{D}_n.\mathbf{s}$
14. $\mathcal{L}_n.\eta = \mathcal{D}_n.\eta$
15. **end**
16. $prune = \mathcal{P}_{i-1}$,
17. **end**
18. **end**
end

Figure 4.20. SESD detection by the z-th thread for N_s signals. This kernel is called from Kernel-Launcher in Fig. 4.19.

4.4.2 Performance Results

The complexity of tree-search detectors is commonly measured in number of expanded nodes, since this allows a fair comparison among different algorithms. Considering a constellation of size M, the number of visited nodes for a MLE detector (n_{MLE}) can be easily calculated as follows:

$$n_{MLE} = \sum_{i=1}^{n_T} M^i. \tag{4.13}$$

Let us consider the simulation a 6×6 system using a 16-QAM constellation. Initial radius is set to infinite and $N_c = 1000$ transmitted signals. The number of nodes of MLE detector is $n_{MLE} = 17.895.696.000$ nodes, however for SESD detector is reduced to $n_{SESD} = 82.850.278$ nodes. Table 4.3 shows the execution time of the SESD detector with the MIMO system considered above as a function of the simulated signals (N_c). By comparing Tables 4.2 and 4.3, we can observe that the reduction of expanded nodes is also reflected in the reduction of the execution time regarding to the MLE detector. Furthermore, EEDC calculation allows to reduce considerably the execution time.

As it happens with the ZFSIC detector, the GPU SESD detector does not has the expected performance. The problems with low complexity and non parallel pattern are not suitable for GPU architecture. Moreover, the warp divergence plays an important role in the poor performance. In CUDA, threads are executed in warps of 32 threads, with all threads in the warp executing the same instruction at the same time. However, in the SESD detector different threads in a warp need to cope with different tasks (see steps 8 and 12 if SESD kernel pseudocode in Fig. 4.20). This is called warp divergence. The GPU hardware is not capable of executing if and $else$ statements at the same time. CUDA serializes the different execution paths to generate correct code. For this reason, the OpenMP version obtains better performance (up to 25 times faster than sequential version) than CUDA version.

4.5 Automatic Sphere Decoder Implementation

The *Automatic Sphere Decoder* (ASD) was proposed in [35]. As the Schnorr-Euchner SD, it is a breadth-first algorithm that does not need an initial sphere radius to find the ML solution. This algorithm stores a list of paths ($\mathcal{L}$) during the tree exploration which contains and defines the bound between the explored part and the unexplored parts of the tree. In each iteration it expands only the node with the shorter associated distance, which is removed from the list. When the expanded node is a leaf node,

Table 4.3. Time Execution comparison in miliseconds and speedup (S_P) of SESD detector with different library configurations for a 6×6 system using 16-QAM constellation as a function of the simulated signals (N_c).

		$N_c = 1000$		$N_c = 10000$		$N_c = 100000$	
		Time[ms]	S_P	Time[ms]	S_P	Time[ms]	S_P
OpenMP Threads	U	$9.80 \cdot 10^3$	1.00	$14.98 \cdot 10^3$	1.00	$156.96 \cdot 10^3$	1.00
	1	$7.55 \cdot 10^3$	1.30	$12.07 \cdot 10^3$	1.24	$128.29 \cdot 10^3$	1.22
	2	$4.17 \cdot 10^3$	2.35	$7.59 \cdot 10^3$	1.97	$64.71 \cdot 10^3$	2.43
	4	$2.39 \cdot 10^3$	4.10	$3.51 \cdot 10^3$	4.27	$34.12 \cdot 10^3$	4.60
	8	$2.39 \cdot 10^3$	7.26	$1.79 \cdot 10^3$	8.37	$18.55 \cdot 10^3$	8.46
	12	$1.35 \cdot 10^3$	9.61	$1.23 \cdot 10^3$	12.18	$12.60 \cdot 10^3$	12.46
	24	$1.02 \cdot 10^3$	13.80	$1.03 \cdot 10^3$	14.54	$9.10 \cdot 10^3$	17.25
	32	$0.72 \cdot 10^3$	13.56	$1.22 \cdot 10^3$	12.28	$8.55 \cdot 10^3$	18.36
	48	$0.69 \cdot 10^3$	14.20	$0.84 \cdot 10^3$	17.83	$6.48 \cdot 10^3$	24.22
	GPU	$32.93 \cdot 10^3$	0.30	$9.32 \cdot 10^3$	1.61	$69.75 \cdot 10^3$	2.25
	(U) GPU	$45.79 \cdot 10^3$	0.21	$12.99 \cdot 10^3$	1.15	$96.42 \cdot 10^3$	1.63

the algorithm terminates and returns this path as the ML solution.

This algorithm has various disadvantages. The main one is the need for a variable-size list of candidate solutions; as shown later can be a drawback for some hardware implementations. Another major disadvantage of this method is that the size of the list can grow too much as the algorithm progresses. Thus, the proposed algorithm uses the following techniques to reducing the size of the list and hence the computational cost:

- Initial radius: The nodes inside the list are purged in each iteration using an initial radius. The radius used can be the distance provided by a suboptimal and low-complexity detection algorithm such as Zero Forcing SIC presented in section 2.3.1:

$$r = \|\mathbf{y} - \mathbf{H}\hat{\mathbf{s}}_{\mathrm{ZF}}\|^2. \tag{4.14}$$

- Reordering channel matrix: A new order of the channel matrix is used in order to discard the largest number of nodes at higher levels and try to keep constant as much as possible the size of the list.

The MIMOPack ASD detector allows to use a Column-Norm-Based Ordering, which orders the columns of the matrix **H** by positioning the columns with lowest 2-norm at first places. It suffices to make a QR decomposition with pivoting, making zeros in the columns in the increasing order of their 2-norm.

Algorithm represented in Fig. 4.25 shows the pseudocode of the ASD method. The detector starts initializing the list with the root node, to which is associated an accumulated ED equal to zero. In the higher level, M nodes are expanded (see Fig.4.21) and stored in the list with the explored paths (in blue). In each iteration, the ASD selects and expands the path (in red in Fig.4.22) within the list with the smallest cumulative Euclidean distance (*minpath*).

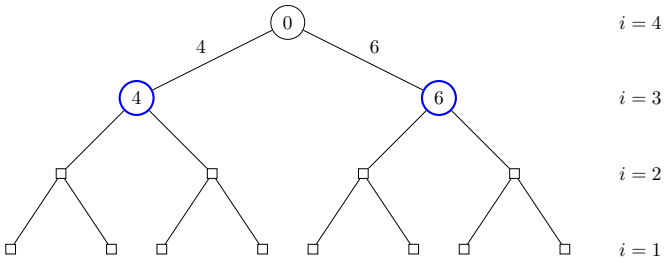


Figure 4.21. Decoding tree of a ASD 3×3 MIMO detector with BPSK symbols: expansion of M nodes in the first level.

Additionally, the algorithm prunes those paths with cumulative ED larger than r. This operation is done by the function *purget_tree*() in step 11, which returns a list with the survivors paths. Then, the expanded branch is replaced by its M children nodes and removed from the list. The level of this new nodes will be equal to $minpath.l - 1$. The same process continues (see Fig. 4.23) until the node to expand in the current iteration is a leaf node (i.e $minpath.l = 1$). At the end of the algorithm *minpath* contains the lattice point that minimizes (4.7) (filled in red in Fig. 4.24).

The ASD is another detector difficult to parallelize, since at each iteration the algorithm must select the minimum path and prune the tree. However, the expansion of the M child nodes of the minimum path can be done in parallel (see steps 2 and 13). As occurs in the MLE detector when N_c is sufficiently large, the best option is to parallelize the loop and not

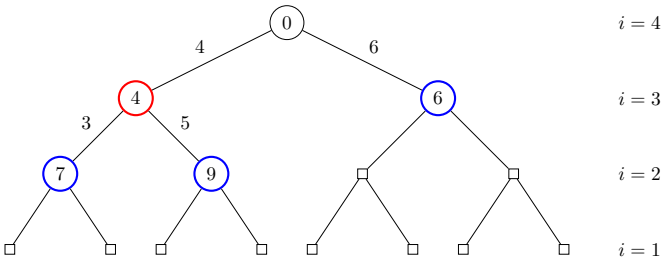


Figure 4.22. Decoding tree of a ASD 3×3 MIMO detector with BPSK symbols: selection and expansion of the node with the smallest cumulative Euclidean distance in the first iteration.

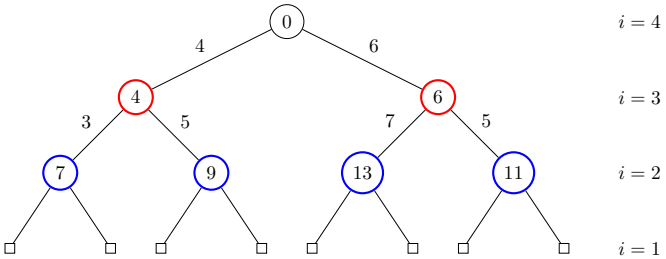


Figure 4.23. Decoding tree of a ASD 3×3 MIMO detector with BPSK symbols: selection and expansion of the node with the smallest cumulative Euclidean distance in the second iteration.

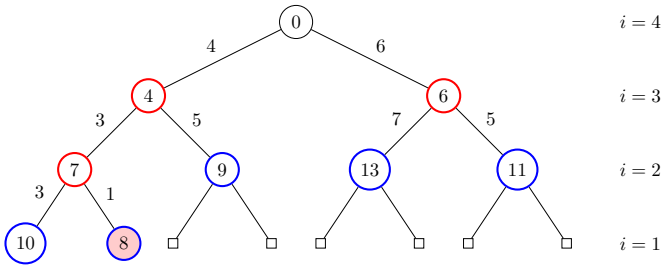


Figure 4.24. Decoding tree of a ASD 3×3 MIMO detector with BPSK symbols: the detection is completed when the node to expand in the current iteration is a leaf node.

Automatic SD PSEUDOCODE

function $[path] = \text{dt\_c\_asd}(\mathbf{R}, \tilde{\mathbf{y}}, r)$
1. $\mathcal{L} = \emptyset$
2. **#pragma omp parallel for**
3. **for** $q = 1 : M$
4. $path.l = n_T$
5. $path.s_{n_T} = \mathcal{O}_q$
6. $path.\eta = |\tilde{y}_i - R_{n_T,n_T} path.s_{n_T}|^2$
7. $\mathcal{L} = \mathcal{L} \cup path$
8. **end**
9. $minpath = min(\mathcal{L}, 1)$
10. $\mathcal{L} = purge\_tree(\mathcal{L}, r) - \{minpath\}$
11. **while** $minpath.l > 1$
12. $i = minpath.l - 1$
13. **#pragma omp parallel for**
14. **for** $q = 1 : M$
15. $path.\mathbf{s} = minpath.\mathbf{s}$
16. $path.l = i$
17. $path.s_i = \mathcal{O}_q$
18. $path.\eta = minpath.\eta + |\tilde{y}_i - \sum_{j=i}^{n_T} R_{i,j} path.s_j|^2$
19. $\mathcal{L} = \mathcal{L} \cup path$
20. **end**
21. $minpath = min(\mathcal{L}, 1)$
22. **if** $minpath.l > 1$
23. $\mathcal{L} = purge\_tree(\mathcal{L}, r) - \{minpath\}$
24. **end**
25. **end**
26. $path = minpath$

end

Figure 4.25. Automatic Sphere Decoder Pseudocode.

the ASD internally. These decisions are transparent to the user and they are performed by the library before the detection.

4.5.1 CUDA Implementation

As mentioned, the main disadvantage of the ASD detector is the need for a variable-size list of candidate solutions, which can be a drawback for some hardware implementations. Specially for GPU implementation, the variable-size of lists and the inherent dependency between the levels of the tree require barrier synchronization among threads that makes very difficult its CUDA implementation. Fortunately, the ASD parallelization can be done with the framework presented in section 4.8 dividing the tree in multiple subtress that can be independently processed by multiple CUDA threads.

4.5.2 Performance Results

To evaluate the performance of the proposed ASD OpenMP detector, the time execution for a 6×6 system and a 16-QAM constellation varying the number od transmitted signals with different number of threads are compared in Table 4.4. Comparing the speedup results with those obtained with the MLE and SESD schemes (tables 4.2 and 4.3, respectively), it can be observe that OpenMP ASD does not obtain as good results when the number of signals is small, mainly caused by the need to maintain large lists for each thread and failures in cache derived from them. However, the algorithm is scalable when the number of signals increases.

The performance of the ASD can be improved using the preprocessing techniques presented in previous section. To validate them, the execution times of four ASD configurations are illustrated in Figure 4.26: Pure ASD detector without initial radius nor matrix channel ordering (ASD), using an initial radius provided by the Zero Forcing SIC detector (ASD-R), using the Column-Norm-Based Ordering (ASD-O) and using both radius and ordering techniques at the same time (ASD-RO).

These techniques are simple and fast, allowing not only to accelerate the detection but also to reduce the size of the node list. In Table 4.5, the effect of the preprocessing techniques on the number and sizes of lists created during the detection has been investigated. It can be observe that the use of the combined ZFSIC radius and CBO ordering allows to reduce

Table 4.4. Time Execution comparison in miliseconds and speedup (S_P) of ASD detector with different library configurations for a 6×6 system using 16-QAM constellation as a function of the simulated signals (N_c).

		$N_c = 1000$		$N_c = 10000$		$N_c = 100000$	
		Time[ms]	S_P	Time[ms]	S_P	Time[ms]	S_P
	U	$0.45 \cdot 10^3$	1.00	$2.94 \cdot 10^3$	1.00	$121.01 \cdot 10^3$	1.00
OpenMP Threads	**1**	$0.43 \cdot 10^3$	1.05	$2.84 \cdot 10^3$	1.04	$119.37 \cdot 10^3$	1.01
	2	$0.25 \cdot 10^3$	1.80	$1.79 \cdot 10^3$	1.64	$66.94 \cdot 10^3$	1.81
	4	$0.13 \cdot 10^3$	3.46	$0.89 \cdot 10^3$	3.30	$36.67 \cdot 10^3$	3.30
	8	$0.08 \cdot 10^3$	5.63	$0.44 \cdot 10^3$	6.68	$20.22 \cdot 10^3$	5.98
	12	$0.07 \cdot 10^3$	6.43	$0.34 \cdot 10^3$	8.65	$15.40 \cdot 10^3$	7.86
	24	$0.08 \cdot 10^3$	5.63	$0.30 \cdot 10^3$	9.80	$10.84 \cdot 10^3$	11.16
	32	$0.08 \cdot 10^3$	5.63	$0.24 \cdot 10^3$	12.25	$8.57 \cdot 10^3$	14.12
	48	$0.12 \cdot 10^3$	3.75	$0.28 \cdot 10^3$	10.50	$7.29 \cdot 10^3$	16.60

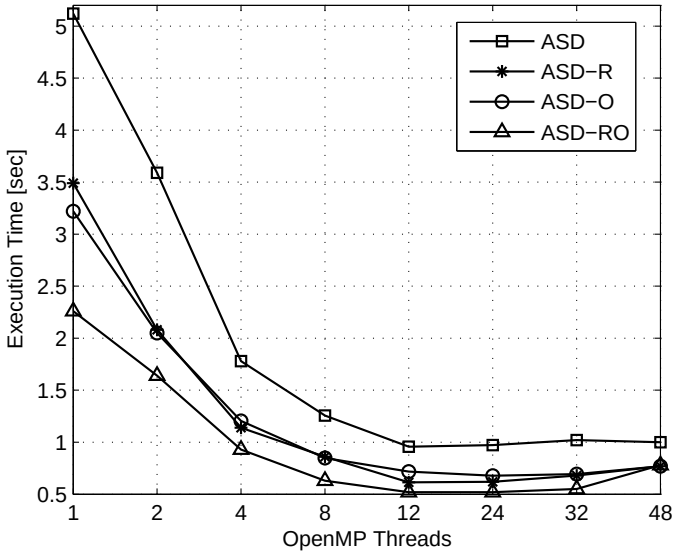


Figure 4.26. Time Execution comparison in seconds of ASD with different detector settings, **ASD-R**, **ASD-O**, and **ASD-RO** as a function of the number of OpenMP threads.

the number of list up to 28% and the average list size up to 40%. This optimization can be noted in Fig. 4.26, where the ASD-RO is able to detect the signals 2.26 times faster than classical ASD.

Table 4.5. Number of lists created, maximum and average list sizes for the **ASD** with different library detector configurations, **ASD-R**, **ASD-O**, and **ASD-RO**.

	ASD	**ASD-R**	**ASD-O**	**ASD-RO**
Number of lists created	222038	222038	160106	159945
Maximum list size	23686	19542	22846	21633
Avg. list size	2447	1635	2038	1448

4.6 K-Best Tree-Search Implementation

The *K-Best Sphere Decoder* (KBEST) is a Fixed-Complexity SD that expands the detection tree from top to bottom and considers only those K survivor candidates with the smallest accumulated Euclidean distance at each level of the tree.

A technique, which allows to increase the performance in terms of Bit Error Rate of this algorithm, is to apply a previous stage called Full Expansion (FE), where in the first n_E levels all the possible values of the constellation for each survivor path are assigned to the symbol at the current level.

The initial list of paths ($\mathcal{L}$) contains the M^{n_E} candidates computed in the Full Expansion stage (see Fig. 4.27). Once the PED of each path has been computed with, the PED values are sorted in ascending order and the K paths having the minimum PED values are stored in the list $\mathcal{D}$. To select the K survivors, function $min(\mathcal{L}, n_K)$ is called with $n_K = K$.

For each survivor path (nodes framed in red in Fig. 4.28), all the possible values of the constellation are assigned to the symbol at the current level and its accumulated distance is calculated (see steps 8-13 of Fig. 4.29). This process is repeated until the lowest level of the tree is reached. The detected signal vector $\hat{\mathbf{s}}$ is given by the path from the root up to the leaf node with the smallest accumulated ED (step 17).

Fully Expansion pseudocode

function $[\mathcal{L}] = \text{fully\_expansion}(\mathbf{R}, \tilde{\mathbf{y}}, n_E)$
1. $\mathcal{L} = \emptyset$
2. **#pragma omp parallel for**
3. **for** $q = 1 : M^{n_E}$
4. $\quad path.\mathbf{s}_{n_T - n_E + 1:n_T} = tree\_path(q, \mathcal{O}, n_E)$
5. $\quad path.\eta = \sum_{i=n_T}^{n_T - n_E + 1} \left| \tilde{y}_i - \sum_{j=i}^{n_T} R_{i,j} path.s_j \right|^2$
6. $\quad \mathcal{L} = \mathcal{L} \cup path$
7. **end**

end

Figure 4.27. Fully Expansion Stage Pseudocode.

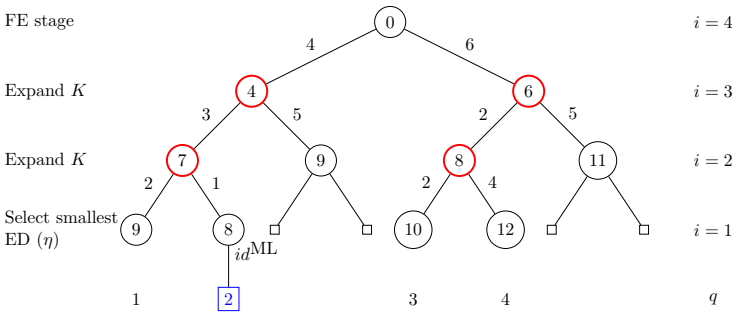


Figure 4.28. Decoding tree of the K-Best algorithm for a 3×3 MIMO system with $K = 2$ and BPSK constellation.

The main advantage of this method is that its maximum number of paths is limited, yielding a fixed computational effort. In addition, the complexity and memory requirements match at every level. This parallelism among detection levels allows an easier hardware implementation of the algorithm.

The fully expansion stage can be easily parallelized keeping the pragma in Fig. 4.27 step 2. Additionally, the K-expansion can also be parallelized since the $K \cdot M$ branches can be explored independently (see steps 5 and 7 in Fig. 4.29).

K-Best SD pseudocode

function $[path] = \text{dt\_kbest}(\mathbf{R}, \tilde{\mathbf{y}}, n_E, K)$
1. $[\mathcal{L}] = fully\_expansion(\mathbf{R}, \tilde{\mathbf{y}}, n_E)$
2. $[\mathcal{D}] = min(\mathcal{L}, K)$
3. $\mathcal{L} = \emptyset$
4. **for** $i = n_T - n_E : -1 : 1$
5. **#pragma omp parallel for**
6. **for** $k = 1 : K$
7. **#pragma omp parallel for**
8. **for** $q = 1 : M$
9. $path.\mathbf{s} = \mathcal{D}_k.\mathbf{s}$
10. $path.s_i = \mathcal{O}_q$
11. $path.\eta = \mathcal{D}_k.\eta + \sum_{i=n_T-n_E}^{1} \left| \tilde{y}_i - \sum_{j=i}^{n_T} R_{i,j} path.s_j \right|^2$
12. $\mathcal{L} = \mathcal{L} \cup path$
13. **end**
14. **end**
15. $[\mathcal{D}] = min(\mathcal{L}, K)$
16. **end**
17. $path = min(\mathcal{D}, 1)$

end

Figure 4.29. K-Best Fixed Complexity Sphere Decoder Pseudocode.

4.6.1 CUDA Implementation

The proposed K-BEST GPU implementation is composed by two kernels. The first one is responsible to calculate the accumulated Euclidean distance for the first n_E levels in the Full Expansion stage (see pseudocode in Fig. 4.30). The calculation of the ED in each branch of the tree is assigned to a different thread. As can be seen in Algorithm 4.31, when all threads finish their calculations in the FE stage, the list of paths $\mathcal{L}$ (distances and symbols) is sent to the CPU to obtain the K survivors. The inherent dependency between the levels of the tree requires synchronization barriers among threads. Therefore, the second kernel includes only the calculation of the cumulative distance for the new $K \cdot M$ branches in the remaining levels and does not carry out the sorting and calculation of the K-best survivors; this process is done sequentially by the CPU (see pseudocode in Fig. 4.32).

KERNEL Fully Expansion Pseudocode

function $[\mathcal{L}]$ = kernel_fully_expansion($\mathbf{T}$, $\tilde{\mathbf{y}}$, n_E)
1. Calculate using the thread global index z:
2. · Time slot identifier $n \in \{1, ..., N_s\}$
3. · Tree-Path identifier $q \in \{1, ..., M^{n_E}\}$
4. **if** $n < N_s$
5. $\mathcal{L}_{n,q}.\mathbf{s}_{n_T-n_E+1:n_T} = tree\_path(q, n_E)$
6. $\mathcal{L}_{n,q}.\eta = \sum_{i=n_T}^{n_T-n_E+1} \left| \tilde{y}_{i,n} - \sum_{j=i}^{n_T} T_{\mathcal{L}_{n,q}.s_j,\Gamma_{i,j}} \right|^2$
7. **end**

end

Figure 4.30. Calculation of one of the branches of the Fully Expansion stage by the z-th thread for N_s signals. This kernel is called from Kernel-Launcher in Fig. 4.31.

4.6.2 Performance Results

The evaluation ot he KBEST implementations has been carried out varying the following parameters: the number of survivors (K), the number of signals transmitted (N_c) and the number of levels to be fully expanded in the FE stage (n_E). The MIMO system considered is a 6×6 system with 16-QAM symbols.

Table 4.6 shows the execution time as a function of the number of survivors K. In all cases, OpenMP outperforms the CUDA performance, which fails to accelerate the sequential version. The main problem of the CUDA code is the dependency between the levels of the tree of the K-BEST detector. The K best survivor's calculation cannot start until all threads have finished the previous GPU computation. In the same way, it cannot proceed with the following branches of the tree until the CPU has not determined the new K survivors and sent such information back to the GPU. The data transfer time on each level (see steps 12 and 14 in Fig. 4.31) of the tree worsens the performance of the GPU version, specially when the size of the constellation and the number of survivors K increase.

However, the impact of this problem disappears gradually when the

K-Best SD Kernel-Launcher Pseudocode

INPUT: $\mathbf{T} \in \mathbb{C}^{M \times n_v}$, $\tilde{\mathbf{y}} \in \mathbb{C}^{n_R \times N_s}$, n_E, K
OUTPUT: $\mathcal{F} \in \mathbb{P}^{N_s}$

function $[\mathcal{F}]$ = dt_cu_kbest($\mathbf{T}$, $\tilde{\mathbf{y}}$, n_E, K)
1. Allocate and copy $\mathbf{T}$, $\tilde{\mathbf{y}}$ in GPU GM
2. Allocate and copy output $\mathcal{L} \in \mathbb{P}^{N_s \times M^{n_E}}$ in GPU GM
3. Select block and grid configuration with $n_{th} = N_s \cdot M^{n_E}$
4. $[\mathcal{L}]$ = kernel_fully_expansion($\mathbf{T}$, $\tilde{\mathbf{y}}$)
5. Copy $\mathcal{L}$ in CPU
6. **for** $n = 1 : N_s$
7. $\quad [\mathcal{D}_{n,:}] = min(\mathcal{L}_{n,:}, K)$
8. **end**
9. Reallocate output $\mathcal{L} \in \mathbb{P}^{N_s \times K \cdot M}$ in GPU GM
10. Select block and grid configuration with $n_{th} = N_s \cdot M \cdot K$
11. **for** $i = n_T - n_E : -1 : 1$
12. $\quad$ Copy $\mathcal{D}$ in GPU Global Memory
13. $\quad [\mathcal{L}]$ = kernel_kbest($\mathbf{T}$, $\tilde{\mathbf{y}}$, $\mathcal{D}$, i)
14. $\quad$ Copy $\mathcal{L}$ in CPU
15. $\quad$ Compute $[\mathcal{D}]$ such as steps 6-8
16. **end**
17. **for** $n = 1 : N_s$
18. $\quad \mathcal{F}_i = min(\mathcal{D}_{n,:}, 1)$
19. **end**

end

Figure 4.31. K-Best Sphere Decoder Kernel-launcher for N_s signals. This launcher calls kernels in Figures 4.30 and 4.32.

complexity of the detection stage increases, for example with the number of tranmistted signals (N_c) in Table 4.7. But specially when the number of levels in the FE stage (see Table 4.8) increases, because this part is very paralelizable and has more computational burden, which helps to improve the performance.

KERNEL KBest Pseudocode

function $[\mathcal{L}]$ = kernel_kbest($\mathbf{T}$, $\tilde{\mathbf{y}}$, $\mathcal{D}$, i)
1. Calculate using the thread global index z:
2. · Time slot identifier $n \in \{1, ..., N_s\}$
3. · Constellation identifier $q \in \{1, ..., M\}$
4. · Survivor identifier $k \in \{1, ..., K\}$
5. **if** $n < N_s$
6. $q' = k \cdot M + q$,
7. $\mathcal{L}_{n,q'}.\mathbf{s} = \mathcal{D}_{n,k}.\mathbf{s}$
8. $\mathcal{L}_{n,q'}.\mathbf{s}_i = q$
9. $\mathcal{L}_{n,q'}.\eta = \mathcal{D}_{n,k}.\eta + |\tilde{y}_{i,n} - \sum_{j=i}^{n_T} T_{\mathcal{L}_{n,q'}.\mathbf{s}_j,\Gamma_{i,j}}|^2$
10. **end**
end

Figure 4.32. Calculation of one of the branches of the K survivors expansion stage by the z-th thread for N_s time instants. This kernel is called from Kernel-Launcher in Fig. 4.31.

Table 4.6. Time Execution comparison in miliseconds and speedup (S_P) of KBEST detector with different library configurations for a 6×6 system using 16-QAM constellation with $N_c = 1000$ and $n_E = 1$ as a function of the number of survivors K.

		$K = 2$		$K = 4$		$K = 10$	
		Time[ms]	S_P	Time[ms]	S_P	Time[ms]	S_P
	U	10.14	1.00	19.63	1.00	51.19	1.00
OpenMP Threads	1	7.38	1.37	14.33	1.37	34.90	1.47
	2	6.61	1.53	11.57	1.70	21.98	2.33
	4	4.78	2.12	4.96	3.96	15.74	3.25
	8	2.96	3.43	4.92	3.99	11.62	4.41
	12	7.19	1.41	4.82	4.07	7.43	6.89
	24	6.87	1.48	5.94	3.30	18.16	2.82
	32	7.86	1.29	6.37	3.08	10.15	5.04
	48	41.22	0.25	65.25	0.30	49.13	1.04
	GPU	317.9	0.03	309.22	0.06	329.11	0.16
	(U) GPU	317.92	0.03	322.1	0.06	333.32	0.15

Table 4.7. Time Execution comparison in miliseconds and speedup (S_P) of KBEST detector with different library configurations for a 6×6 system using 16-QAM constellation with $n_E = 1$ and $K = 10$ as a function of the simulated signals (N_c).

		$N_c = 1000$		$N_c = 10000$		$N_c = 100000$	
		Time[ms]	S_P	Time[ms]	S_P	Time[ms]	S_P
	U	51.19	1.00	$4.14 \cdot 10^2$	1.00	$37.85 \cdot 10^2$	1.00
OpenMP Threads	1	34.90	1.47	$3.24 \cdot 10^2$	1.28	$29.37 \cdot 10^2$	1.29
	2	21.98	2.33	$1.96 \cdot 10^2$	2.11	$15.61 \cdot 10^2$	2.42
	4	15.74	3.25	$1.00 \cdot 10^2$	4.14	$8.03 \cdot 10^2$	4.71
	8	11.62	4.41	$0.93 \cdot 10^2$	4.45	$4.45 \cdot 10^2$	8.51
	12	7.43	6.89	$0.54 \cdot 10^2$	7.67	$6.12 \cdot 10^2$	6.18
	24	18.16	2.82	$0.34 \cdot 10^2$	12.18	$2.81 \cdot 10^2$	13.47
	32	10.15	5.04	$1.07 \cdot 10^2$	3.87	$10.25 \cdot 10^2$	3.69
	48	49.13	1.04	$1.08 \cdot 10^2$	3.83	$2.88 \cdot 10^2$	13.14
	GPU	329.11	0.16	$5.64 \cdot 10^2$	0.73	$26.32 \cdot 10^2$	1.44
	(U) GPU	333.32	0.15	$5.78 \cdot 10^2$	0.72	$26.58 \cdot 10^2$	1.42

4.7 Hard-Output Fixed-Complexity Sphere Decoder

In [39], a method called *Fixed-Complexity Sphere Decoder* (FSD) was proposed in order to address simultaneously performance and fixed computation aspects. Algorithm 4.34 shows the pseudocode of FSD algorithm, which combines a preprocessing stage with a tree-search composed of two stages:

- Full Expansion (FE): In the first n_E levels, all the possible values of the constellation for each survivor path are assigned to the symbol at the current level (coloured in red in Fig. 4.35).
- Single-Path Expansion (SE): The SE stage starts from each retained path and obtains the remaining unknows, those coloured in blue in the lowest $n_T - n_E$ tree-levels (see 4.35) using Succesive Interference Cancellation (SIC) (see section 2.3.1).

The symbols are detected following a specific ordering also proposed by the authors in [39]. As it was shown in [77], the maximum detection diversity can be achieved with the FSD if the following value of n_E is chosen:

Table 4.8. Time Execution comparison in miliseconds and speedup (S_P) of KBEST detector with different library configurations for a 6×6 system using 16-QAM constellation with $N_c = 100000$ and $K = 10$ as a function of the number of levels n_E to be expanded in the FE stage.

		$n_E = 2$		$n_E = 3$		$n_E = 4$	
		Time[ms]	S_P	Time[ms]	S_P	Time[ms]	S_P
	U	$4.86 \cdot 10^3$	1.00	$36.14 \cdot 10^3$	1.00	$766.69 \cdot 10^3$	1.00
OpenMP Threads	1	$3.81 \cdot 10^3$	1.28	$30.64 \cdot 10^3$	1.18	$625.79 \cdot 10^3$	1.23
	2	$2.36 \cdot 10^3$	2.06	$16.24 \cdot 10^3$	2.23	$316.90 \cdot 10^3$	2.42
	4	$1.21 \cdot 10^3$	4.02	$8.37 \cdot 10^3$	4.32	$172.69 \cdot 10^3$	4.44
	8	$0.56 \cdot 10^3$	8.68	$4.39 \cdot 10^3$	8.23	$92.11 \cdot 10^3$	8.32
	12	$0.52 \cdot 10^3$	9.35	$2.98 \cdot 10^3$	12.13	$61.89 \cdot 10^3$	12.39
	24	$0.34 \cdot 10^3$	14.29	$2.50 \cdot 10^3$	14.46	$44.77 \cdot 10^3$	17.13
	32	$0.28 \cdot 10^3$	17.36	$1.90 \cdot 10^3$	19.02	$37.10 \cdot 10^3$	20.67
	48	$0.47 \cdot 10^3$	10.34	$1.76 \cdot 10^3$	20.53	$26.66 \cdot 10^3$	28.76
GPU	GPU	$2.73 \cdot 10^3$	1.78	$11.17 \cdot 10^3$	3.24	$88.95 \cdot 10^3$	8.62
	(U) GPU	$2.78 \cdot 10^3$	1.75	$11.94 \cdot 10^3$	3.03	$88.86 \cdot 10^3$	8.63

$$n_E \geq \sqrt{n_T} - 1. \tag{4.15}$$

The preprocessing stage (see Fig. 4.1) orders the columns of the channel matrix $\mathbf{H}$ in order to place the signals that suffers the largest noise amplification in the first n_E levels, since no candidate is being discarded at this level and the final performance will not be altered. In the same way, the signals that suffer the smallest noise amplification will be placed at SE levels. The steps to carry out for each column ($i = n_T, \ldots, 1$) are the followings:

- The pseudoinverse $\mathbf{H}_i^{\dagger} = (\mathbf{H}_i^H \mathbf{H}_i)^{-1} \mathbf{H}_i^H$ is computed, where $(\mathbf{H}_i)$ is the matrix formed by the columns of the indices not selected in previous iterations.
- The column index j is selected according to:

$$j = \begin{cases} \arg\max \|\mathbf{H}_i^{\dagger}\|^2, & \text{if a symbol for FE is searched} \\ \arg\min \|\mathbf{H}_i^{\dagger}\|^2, & \text{otherwise.} \end{cases} \tag{4.16}$$

This preprocessing needs a high computational cost since requires the computation of a pseudoinverse of matrix $\mathbf{H}_i^{\dagger} = (\mathbf{H}_i^H \mathbf{H}_i)^{-1}\mathbf{H}_i^H$ in every step. By using the QR decomposition we can to perform the pseudoinverse by solving one linear system that is next presented. Let us call

$$\mathbf{H}_i^{\dagger} = \underbrace{(\mathbf{H}_i^H \mathbf{H}_i)}_{\mathbf{G}}{}^{-1}\mathbf{H}_i^H. \tag{4.17}$$

If we perform the decomposition of $\mathbf{H}_i$ such as $\mathbf{H}_i = \mathbf{Q}_i\mathbf{R}_i$ (see Fig. 4.33), matrix $\mathbf{G}$ can be expressed as:

$$\mathbf{G} = ((\mathbf{Q}_i\mathbf{R}_i)^H(\mathbf{Q}_i\mathbf{R}_i))^{-1} = (\mathbf{R}_i^H\mathbf{Q}_i^H\mathbf{Q}_i\mathbf{R}_i)^{-1} = (\mathbf{R}_i^H\mathbf{I}\mathbf{R}_i)^{-1} = \tilde{\mathbf{R}}_i^{-1}\tilde{\mathbf{R}}_i^{-H}. \tag{4.18}$$

As $\mathbf{H}_i^H = \mathbf{R}_i^H\mathbf{Q}_i^H = [\tilde{\mathbf{R}}_i^H \ 0]\mathbf{Q}_i^H$, the right hand side of the equation (4.17) can be expressed as:

$$\mathbf{G}\mathbf{H}_i^H = \tilde{\mathbf{R}}_i^{-1}\tilde{\mathbf{R}}_i^{-H}[\tilde{\mathbf{R}}_i^H \ 0]\mathbf{Q}_i^H = \tilde{\mathbf{R}}_i^{-1}[\mathbf{I} \ 0]\mathbf{Q}_i^H = \tilde{\mathbf{R}}_i^{-1}\tilde{\mathbf{Q}}_i^{(1)H} \tag{4.19}$$

where $\tilde{\mathbf{Q}}_i^{(1)H} = [\mathbf{I} \ 0]\tilde{\mathbf{Q}}_i^H$, see Fig. 4.33.

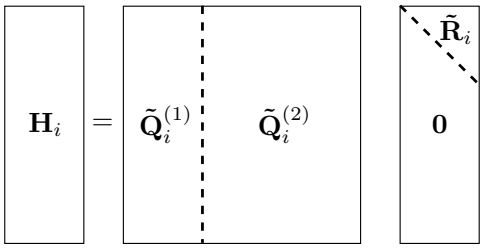


Figure 4.33. QR decomposition of matrix $\mathbf{H}_i$

Now (4.17) can be equivalently expressed as $\mathbf{H}_i^{\dagger} = \tilde{\mathbf{R}}_i^{-1}\tilde{\mathbf{Q}}_i^{(1)H}$ and can obtain $\mathbf{H}_i^{\dagger}$ solving the following upper triangular system:

$$\tilde{\mathbf{R}}_i\mathbf{H}_i^{\dagger} = \tilde{\mathbf{Q}}_i^{(1)H}. \tag{4.20}$$

Hard Fixed SD pseudocode

function $[path] = \text{dt\_c\_hfsd}(\mathbf{R}, \tilde{\mathbf{y}}, n_E)$
1. $[\mathcal{L}] = fully\_expansion(\mathbf{R}, \tilde{\mathbf{y}}, n_E)$
2. **#pragma omp parallel for**
3. **for** $q = 1 : M^{n_E}$
4. **for** $i = n_T - n_E : -1 : 1$
5. $\mathcal{L}_q.s_i = \mathcal{Q}\left\{\frac{\tilde{y}_i - \sum_{j=i+1}^{n_T} R_{i,j}\mathcal{L}_q.s_j}{R_{i,i}}\right\}$
6. **end**
7. $\mathcal{L}_q.\eta = \mathcal{L}_q.\eta + \sum_{i=n_T-n_E}^{1} \left|\tilde{y}_i - \sum_{j=i}^{n_T} R_{i,j}\mathcal{L}_q.s_j\right|^2$
8. **end**
9. $path = min(\mathcal{L}, 1)$

end

Figure 4.34. Hard Fixed Complexity Sphere Decoder Pseudocode.

Note that, the M^{n_E} independent branches of the hard-output FSD can be simultaneously executed by different threads using the pragma in Fig. 4.34 step 2.

4.7.1 CUDA Implementation

Algorithm described in Fig. 4.36 shows the steps needed to launch the kernel. The proposed FSD GPU implementation is composed by one single kernel that is responsible for calculating the accumulated Euclidean distance for the first n_E levels in the Full Expansion stage and also obtains the remaining unknows (those in the lowest $n_T - n_E$ tree-levels) using SIC. Then, the M^{n_E} independent branches of the hard-output FSD scheme of all signals are simultaneously executed by different threads. The output of the kernel is a matrix list $\mathcal{L}$ which contains the M^{n_E} explored paths for each transmitted signal. This matrix is copied to the CPU memory, which calculates the path with the minimum distance in order to obtain the final solution.

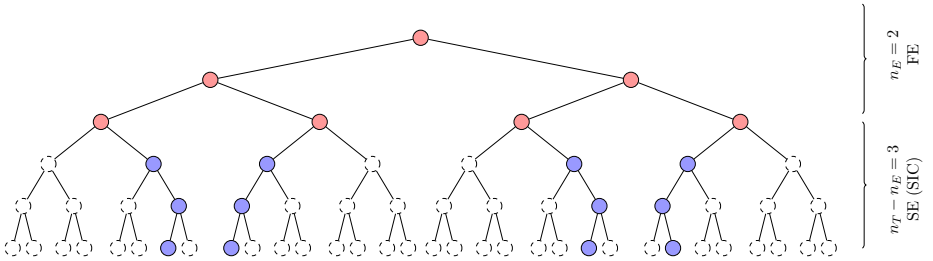


Figure 4.35. Decoding tree of the FSD algorithm for a 5×5 MIMO system with $n_E = 2$ and QPSK modulation.

Hard Fixed SD Kernel-Launcher Pseudocode

Input: $\mathbf{T} \in \mathbb{C}^{M \times n_v}$, $\mathbf{R} \in \mathbb{C}^{n_T \times n_T}$, $\tilde{\mathbf{y}} \in \mathbb{C}^{n_R \times N_s}$, n_E
Output: $\mathcal{F} \in \mathbb{P}^{N_s}$

function $[\mathcal{F}]$ = dt_cu_hfsd($\mathbf{T}$, $\tilde{\mathbf{y}}$, n_E)
1. Allocate and copy $\mathbf{T}$, $\mathbf{R}$ and $\tilde{\mathbf{y}}$ in GPU GM
2. Allocate output data $\mathcal{L} \in \mathbb{P}^{N_s \times M^{n_E}}$ in GPU GM
3. Copy constellation symbols $\mathcal{O}$ in GPU CM
3. Select block and grid configuration with $n_{th} = N_s \cdot M^{n_E}$
4. $[\mathcal{L}]$ = kernel_hfsd($\mathbf{T}$, $\mathbf{R}$, $\tilde{\mathbf{y}}$)
5. Copy $\mathcal{L}$ in CPU
6. **for** $n = 1 : N_s$
7. $\quad \mathcal{F}_n = min(\mathcal{L}_{n,:}, 1)$
8. **end**

end

Figure 4.36. Hard Fixed Complexity Sphere Decoder Kernel-launcher for N_s signals. This launcher calls kernel in Fig. 4.37.

4.7.2 Performance Results

Table 4.9 collects the execution time values of the FSD implementation in miliseconds for a 6×6 system and a 16-QAM constellation as a function of the simulated signals (N_c).

As occurs with the ZFSIC detector, the use of the EEDC does not achieve the expected performance. This is more noticiable in the case of GPU implementation whose time execution is increased. The FSD algo-

KERNEL Hard Fixed SD Pseudocode

function $[\mathcal{L}]$ = kernel_hfsd($\mathbf{T}$, $\mathbf{R}$, $\tilde{\mathbf{y}}$, n_E)
1. Calculate using the thread global index z:
2. · Time slot identifier $n \in \{1, ..., N_s\}$
3. · Tree-Path identifier $q \in \{1, ..., M^{n_E}\}$
4. **if** $n < N_s$
5. $\mathcal{L}_{n,q}.\mathbf{s}_{n_T-n_E+1:n_T} = tree\_path(q, n_E)$
6. $\mathcal{L}_{n,q}.\eta = \sum_{i=n_T}^{n_T-n_E+1} \left| \tilde{y}_{i,n} - \sum_{j=i}^{n_T} T_{\mathcal{L}_{n,q}.s_j,\Gamma_{i,j}} \right|^2$
7. **for** $i = n_T - n_E : -1 : 1$
8. $x = \mathcal{Q}\left\{ \frac{\tilde{y}_{i,n} - \sum_{j=i+1}^{n_T} T_{\mathcal{L}_{n,q}.s_j \Gamma_{i,j}}}{R_{i,i}} \right\}$
9. $\mathcal{L}_{n,q}.s_i = \langle x \rangle$
10. $\mathcal{L}_{n,q}.\eta = \mathcal{L}_{n,q}.\eta + |\tilde{y}_{i,n} - \sum_{j=i}^{n_T} T_{\mathcal{L}_{n,q}.s_j,\Gamma_{i,j}}|^2$
11. **end**
12. **end**

end

Figure 4.37. FSD detection by the z-th thread for N_s signals. This kernel is called from Kernel-Launcher in Fig. 4.36.

rithm uses Fully Expansion in the first n_E levels and SIC technique in the remaining $n_T - n_E$ levels. SIC function needs to compare each estimated symbol with each constellation symbol in step 5 in Fig. 4.34. This quantization affects the performance obtained with CUDA version because it creates serious problems related to the warp divergence. For this reason, when the number of levels in the FE stage becomes higher, the number of SIC levels ($n_T - n_E$) decreases and the CUDA version obtains better speedup than OpenMP version (see Fig. 4.38). Here, the number of fully expanded levels is calculated as $n_E \geq \sqrt{n_T} - 1$.

4.8 WinTrees: a Divide-and-Conquer framework for Tree-Search-Based MIMO detectors

An original framework, called WinTrees, has been developed to facilitate the free-design and implementation of tree-based MIMO detectors. This

Table 4.9. Time Execution comparison in miliseconds and speedup (S_P) of FSD detector with different library configurations for a 6×6 system using 16-QAM constellation with $n_E = 2$ as a function of the simulated signals (N_c).

		$N_c = 1000$		$N_c = 10000$		$N_c = 100000$	
		Time[ms]	S_P	Time[ms]	S_P	Time[ms]	S_P
	U	121.67	1.00	1007.71	1.00	9407.4	1.00
OpenMP Threads	1	111.32	1.09	1014.22	0.99	8654.10	1.09
	2	57.07	2.13	566.60	1.78	5445.70	1.73
	4	33.68	3.61	281.80	3.58	2387.30	3.94
	8	17.15	7.09	144.32	6.98	1243.90	7.56
	12	14.42	8.44	99.90	10.09	856.10	10.99
	24	16.64	7.31	89.51	11.26	702.11	13.40
	32	15.48	7.86	71.02	14.19	564.21	16.67
	48	74.84	1.63	109.4	9.21	712.38	13.21
	GPU	308.72	0.39	378.9	2.66	793.05	11.86
	(U) GPU	302.01	0.40	376.1	2.68	760.32	12.37

framework is primarly devised to ease the parallelization of any kind of hard output detector but it can be also used for sequential detector implementations. As the name suggests, WinTrees starts from a detection tree and gets or wins small subtrees following a divide and conquer strategy [78]. Then, the main problem is transformed in several subproblems that can be independently processed (figure 4.39).

Figure 4.40 enumerates the steps done by the framework's kernel in multi-core mode. The first phase generates the subproblems in three steps: Exhaustive Expansion, Pruning Subtrees and Remodel System. In a second phase the framework executes simultaneously the N_{WT} detectors parallelizing the loop (step 7) or not, depending on the user requirements. The last phase takes the final decision using the outputs of all the subproblems. Note that, Exhaustive Expansion stage can also be parallelized with OpenMP (Fig. 4.27) or CUDA (Fig. 4.30). The user can also parallelize the loop in step 7), distributing the N_{WT} subproblems among different CUDA cores. Fig 4.41 shows how list $\mathcal{D}$ can be calculated with the MIMOPack algorithms presented in this chapter.

For example, the detection tree depicted in Fig. 4.39 shows a 5×5

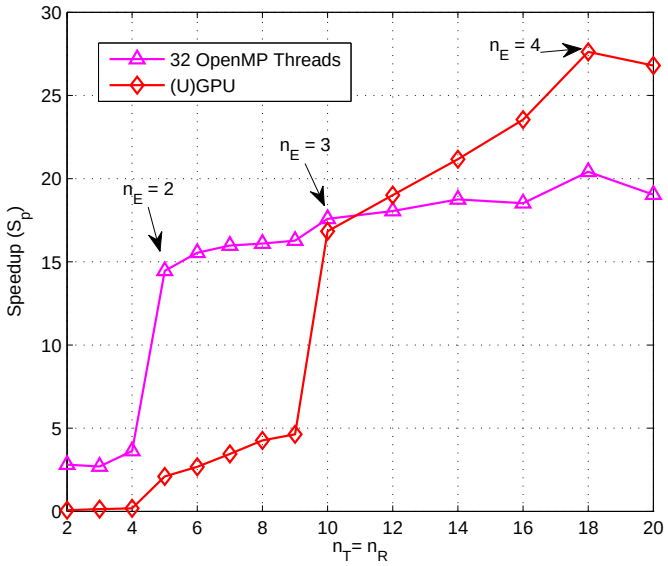


Figure 4.38. Speedup (S_P) comparison of FSD detector with different library configurations for a $n_R \times n_T$ MIMO system with 16-QAM constellation and $N_c = 10000$ signals.

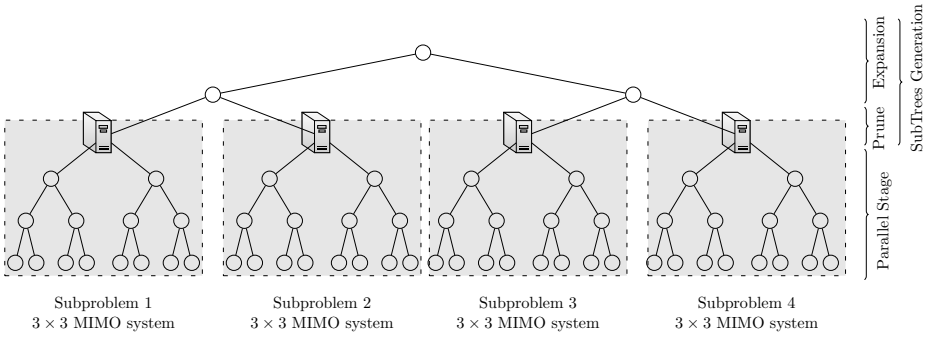


Figure 4.39. WinTrees stages for a detection tree of a 5×5 MIMO system using a BPSK constellation with $n_E = 2$ levels exhaustive expanded.

MIMO system with a BPSK constellation ($M = 2$) that can be split into 4 different MIMO systems of size 3×3 as follows:

C/OpenMP Framework pseudocode for third party detectors

function $[\hat{\mathbf{s}}, \eta]$ = hard_wintrees($\mathbf{R}$, $\tilde{\mathbf{y}}$, n_E [, r] [, K])
1. $\mathcal{D} = \emptyset$
2. $[\mathcal{L}] = fully\_expansion(\mathbf{R}, \tilde{\mathbf{y}}, n_E)$
3. $[\mathcal{L}] = purge\_tree(\mathcal{L}, r)$ or $[\mathcal{L}] = min(\mathcal{L}, K)$
4. $[\breve{\mathbf{R}}, \breve{\mathbf{y}}] = resize\_model(\mathbf{R}, \tilde{\mathbf{y}}, \mathcal{L})$
5. $N_{WT} = length(\mathcal{L})$
6. **#pragma omp parallel for**
7. **for** $n = 1 : N_{WT}$
8. $\quad path = third\_party\_detector(\breve{\mathbf{R}}, \breve{\mathbf{y}}_{n,:})$
9. $\quad path.\mathbf{s}_{n_T - n_E + 1:n_T} = \mathcal{L}_n.\mathbf{s}$
10. $\quad path.\eta = path.\eta + \mathcal{L}_n.\eta$
11. $\quad \mathcal{D} = path \cup \mathcal{D}$
12. **end**
13. $path = min(\mathcal{D}, 1)$
14. $\eta = path.\eta$
15. $\hat{\mathbf{s}} = path.\mathbf{s}$

end

Figure 4.40. Framework Pseudocode for third party detectors.

- Exhaustive Expansion: In the $n_E = 2$ upper levels of the tree, all branches are considered. Thereby, $N_{WT} = M^{n_E} = 4$ partial euclidean distances (PED) are computed, one for each explored branch. For a given branch, the latest n_E components of $\mathbf{s}$ are fixed. After that, the partial euclidean distance (PED) is computed using a submatrix of matrix $\mathbf{R}$ and a subvector of the received vector $\tilde{\mathbf{y}}$ (see Fig. 4.42).

  It should be noted that the parallelism degree achieved by the framework depends directly on the number of exhaustive levels considered (n_E) since it determines the number of subproblems that can be independently computed.

- Pruning Subtrees: The framework incorporates a parameter that allows to reduce the number of generated subproblems to solve. In the framework the following types of pruning are allowed:

  – Radius: The subproblems with the Partial Euclidean Distances

Framework pseudocode for MIMOPack detectors

function $[\hat{\mathbf{s}}, \eta]$ = hard_wintrees($\mathbf{R}$, $\tilde{\mathbf{y}}$, n_E $[, r]$ $[, K]$ $[, det]$)

1. $[\mathcal{L}] = fully\_expansion(\mathbf{R}, \tilde{\mathbf{y}}, n_E)$
2. $[\mathcal{L}] = purge\_tree(\mathcal{L}, r)$ or $[\mathcal{L}] = min(\mathcal{L}, K)$
3. $[\breve{\mathbf{R}}, \breve{\mathbf{y}}] = resize\_model(\mathbf{R}, \tilde{\mathbf{y}}, \mathcal{L})$
4. **if** CUDA
5. $\quad [\mathcal{D}] = dt\_cu\_name\_wrapper(\breve{\mathbf{R}}, \breve{\mathbf{y}}, det)$
6. **else**
7. $\quad [\mathcal{D}] = dt\_c\_name\_wrapper(\breve{\mathbf{R}}, \breve{\mathbf{y}}, det)$
8. **end**
9. **for** $n = 1 : N_{WT}$
10. $\quad \mathcal{D}_n.\mathbf{s}_{n_T - n_E + 1:n_T} = \mathcal{L}_n.\mathbf{s}$
11. $\quad \mathcal{D}_n.\eta = \mathcal{D}_n.\eta + \mathcal{L}_n.\eta$
12. **end**
13. $path = min(\mathcal{D}, 1)$
14. $\eta = path.\eta$
15. $\hat{\mathbf{s}} = path.\mathbf{s}$

end

Figure 4.41. Framework Pseudocode for MIMOPack detectors.

$$\begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \tilde{y}_3 \\ \tilde{y}_4 \\ \tilde{y}_5 \end{pmatrix} - \left(\begin{array}{ccc:cc} R_{11} & R_{12} & R_{13} & R_{14} & R_{15} \\ 0 & R_{22} & R_{23} & R_{24} & R_{25} \\ 0 & 0 & R_{33} & R_{34} & R_{35} \\ \hdashline 0 & 0 & 0 & R_{44} & R_{45} \\ 0 & 0 & 0 & 0 & R_{55} \end{array}\right) \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{pmatrix} \begin{matrix} \\ \\ \} n_T - n_E = 3 \\ \\ \} n_E = 2 \end{matrix}$$

Figure 4.42. Data used in the WinTrees Exhaustive Expansion stage.

greater than a radius (r) provided by the user will be discarded.

– K-Best: Only the K best subproblems will be considered in the

detection process (i.e the K subproblems with the lowest PED). This type of pruning not only allows to reduce the complexity of the detection but also adjust the parallelism degree to the platform where the detector will be executed selecting a suitable value of K.

- Resized System: The exhaustive expansion and pruning stages generates jointly N_{WT} different subproblems, thus, it is necessary to resize the overall detection problem. Now, the $n_T \times n_T$ original MIMO system becomes N_{WT} different $n_T - n_E \times n_T - n_E$ MIMO systems. For the sake of simplicity, henceforth, the subtree size will be named as $n_{ST} = n_T - n_E$. For each subproblem, the received symbol vector $\tilde{\mathbf{y}}$ will be different since is affected by the estimated symbols computed up to now (see Fig. 4.42), thus it is necessary to compute them after the computation of the remaining estimated symbols $i \in \{1, \ldots, n_{ST}\}$ for each subproblem n as:

$$\breve{\mathbf{y}}_{i,n} = \tilde{\mathbf{y}}_i - \sum_{j=n_{ST}+1}^{n_T} R_{i,j} \mathcal{L}_n . s_j . \tag{4.21}$$

Figure 4.43. Data used in the WinTrees Resized System stage.

The stage of subtrees generation, N_{WT} detectors can be launched simultaneously (or not) to proceed with the detection. This step of the framework uniquely needs a portion of the original matrix $\mathbf{R}$, denoted by $\breve{\mathbf{R}}$ (see Fig. 4.44). Through this scheme, the user can run its own detector

or any other provided by the MIMOPack library in the boxes colored in gray in Fig. 4.39.

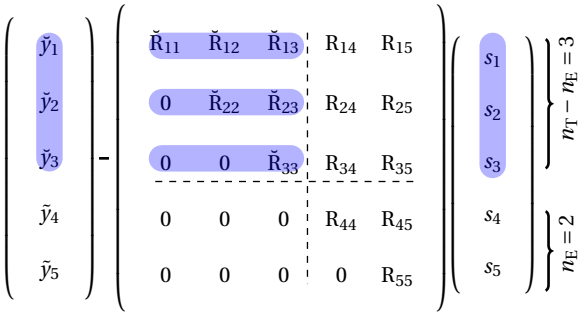


Figure 4.44. Data used by the subproblems generated by the WinTrees Generation Subtrees process.

Once all subproblems have finished, the framework has a list of possible solutions in $\mathcal{D}$. To complete the detection, in step 13, the framework gets the path with the minimum distance.

Observe that the algorithms implemented using this framework have been devised by means of a Divide-and-Conquer (D&C) approach and the theoretical computational cost may suffer a complexity reduction, either in a sequential implementation or in a parallel one. Let us consider, for example, the MLE performance as target for the framework. The sequential MLE algorithm (see Fig. 4.14) exhibits a computational cost of $\Theta(M^{n_T} \cdot n_T^2)$. The corresponding D&C approach would show a sequential computational cost of $\Theta(M^{n_T} \cdot (n_T - n_E)^2)$. Thus, the D&C approach provides a theoretical speedup $S_p = M^{n_T} \cdot (\frac{n_T}{n_T - n_E})^2$, even in a sequential implementation. The theoretical parallel speedup can reach up to $S_p = M^{n_E} \cdot (\frac{n_T}{n_T - n_E})^2$ if M^{n_E} processors were used.

Graphical results represented in Figure 4.45 illustrate the execution time comparison between the MLE sequential demodulator without the use of WinTrees, and the WinTrees parallelization with OpenMP and CUDA MLE demodulators (see Fig. 4.40,for the framework performance). In the OpenMP version, the N_{WT} threads are distributed among all the CPU cores. On the other hand, the CUDA version creates a grid with $n_{th} = N_{WT}$ threads such that a single CUDA core solves one subproblem.

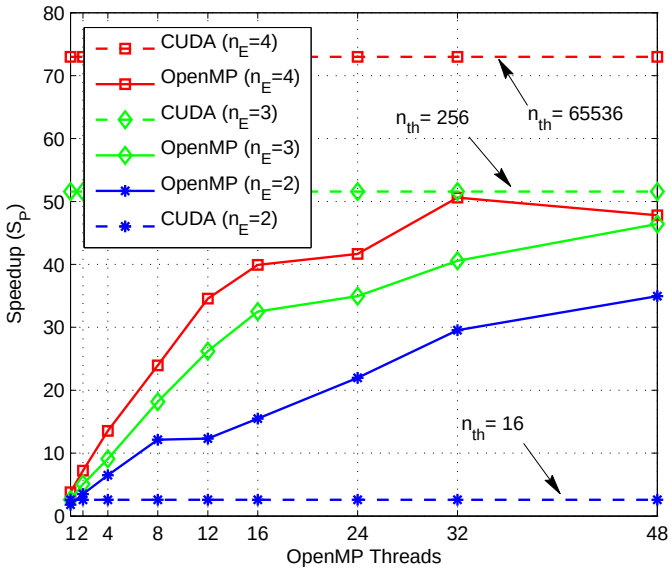


Figure 4.45. Speedup of a MLE detector parallelized using WinTrees framework with different library configurations over a 6×6 MIMO system with 16QAM modulation.

4.9 Conclusions

In this chapter, several Hard-Output MIMO configurable detectors with different complexities have been implemented in OpenMP and GPU. The detection stage is highly accelerated through exploiting two parallelism levels: first, independent parts of the considered algorithms are processed in parallel and, second, the detection step is carried out simultaneously for all signals in the simulation system through forwarding each one to a different thread.

A study of the execution time for different number of signals and system sizes was carried out to evaluate the influence of these parameters on the library performance.

The selected methods were also implemented in a sequential and unoptimized version. The execution times of this implementation were compared to the execution times of the proposed OpenMP and GPU implementations.

Speedup results showed that the multi-core and GPU-based detector performs up to 30 times faster than sequential versions for some cases. Moreover, the speedup increases with the system size and number of transmitted signals, showing the interest of multi-core and GPU implementations for configurations managing many signals simultaneously in very large MIMO systems.

Efficient Euclidean distance calculation reduces $20 - 30\%$ the time needed to calculate the ED in the exploration tree for most detectors. However, for ZFSIC and FSD detectors, the simultaneous use of both matrices $\mathbf{R}$ and $\mathbf{T}$ worsens the performance in some cases. This is mainly due to the extra memory accesses to the matrix $\mathbf{T}$ and to the failures in cache memory for OpenMP versions. For the CUDA versions EEDC optimization is not recommended since the time required to transfer the matrix $\mathbf{T}$ to the GPU memory can be higher than the gain obtained from it use.

Additionally, a divide-and-conquer framework called WinTrees was presented and assessed. WinTrees was designed to assist the parallelization of Hard-Output detectors. Its D&C approach makes possible the reduction of the detection complexity, either in a sequential implementation or in parallel one. Results showed that the speedup increases with the number of subproblems genrated from the original MIMO tree-search, achieving a speedup up to 70 for GPU computing and 45 for the multi-core version.

Future work is needed to decrease the runtime for these Hard-Output detectors. Implementations of the detectors are made in a generic manner to allow their execution on any type of NVIDIA GPU architecture. However, depending on the model installed on the user's platform, some specific features (e.g shared memory, nested parallelism, streams) may be exploited to decrease the execution time. Therefore, the algorithms will be reimplemented wherever possible, in order to optimize them for each architecture. Last, it can be remarked that some information presented in this chapter has been published in international journals. The OpenMP implementation of the FSD has been reported in [79]. The paper containing the design and GPU implementation using shared memory and different GPU features of the FSD was reported in [76].

Implementation of Soft-Output MIMO Detectors

5

Implementation of Soft-Output MIMO Detectors

5

The use of soft detection in MIMO-BICM is necessary to improve the reliability of the MIMO-BICM systems with respect to the use of hard detection. The optimal demodulator is the soft-output *maximum a posteriori probability* (MAP) detector. This detector is the best in terms of BER but needs a high computational power.

In this chapter, we propose a set of efficient soft-output detectors that allow to reduce the computational complexity of a MAP detector. Preliminary results show that our implementations achieve a significant complexity reduction. Furthermore, as seen in the previous chapter, the use of the efficient Euclidean Distance Calculation (EEDC) allows to reduce up to 25% the execution time.

5.1 Introduction

The soft output detection is used in *Bit-Interleaved Coded Modulation* (BICM) MIMO systems [21], where the information bits $\mathbf{b}$ are encoded and optionally interleaved before the transmission [see Chapter 2, Section 2.1.1]. A channel encoder replicates some of the information to be trans-

mitted, so that the receiver can recover the original information even when data losses occur or the channel conditions are not too good. The main task of the soft output MIMO detector is to produce the estimation of the coded and interleaved received bits $\mathbf{c}$ in terms of log-likehood ratios LLRs as:

$$\Lambda_{i,k} = \log \frac{\mathrm{P}(c_{i,k} = 1|\mathbf{y}, \mathbf{H})}{\mathrm{P}(c_{i,k} = 0|\mathbf{y}, \mathbf{H})}, \tag{5.1}$$

for all bits $i = 1, ..., n_T$, $k = 1, \ldots, m$ in the label coded c. The sign of an LLR-value $\Lambda_{i,k}$ indicates whether the corresponding bit $c_{i,k}$ is more likely to be 0 or 1, thus negative values indicates 0 and positive values 1. The magnitude $|\Lambda_{i,k}|$ denotes the reliability of the estimate $c_{i,k} = \mathrm{sign}(\Lambda_{i,k})$. Large magnitudes indicate high confidence, whereas low magnitudes correspond to estimates with low reliability. These LLRs are deinterleaved (if needed) and used by the channel decoder to make final decisions about the transmitted sequence bits $\hat{\mathbf{b}}$.

Figure 5.1 shows the necessary steps to carry out the Soft-Output detection, where most of them are common to the hard-output detection. MIMOPack also allows the execution of the soft output detectors in different execution modes. Figure 5.2 shows the pseudocode for the detector in sequential, multi-core, multi-GPU and heterogeneus modes that are managed identically to that view in chapter 4, section 4.1. The chapter is organized as follows: first the *Maximum A Posteriori Probability* (MAP) and *Max-Log Detectors Implementation* (MLA) are introduced. These detectors exhibit good performance in terms of Bit Error Rate but have a huge computational complexity. For this reason, two suboptimal soft detectors based in a extension of the list provided by the Hard-Ouput FSD were implemented: the *Soft Fixed Complexity Sphere Decoder* (SFSD) and the *Fully Parallel Soft Fixed Sphere Decoder* (FPSD).

Each description contains the C/OpenMP pseudocode and the CUDA algorithm version. In order to make an assessment of the MIMOPack Soft-Output detectors, a performance subsection is presented for each detector, which shows time measurements with different configurations taken on the Computer System A described in Chapter 3, Section 3.1.3.

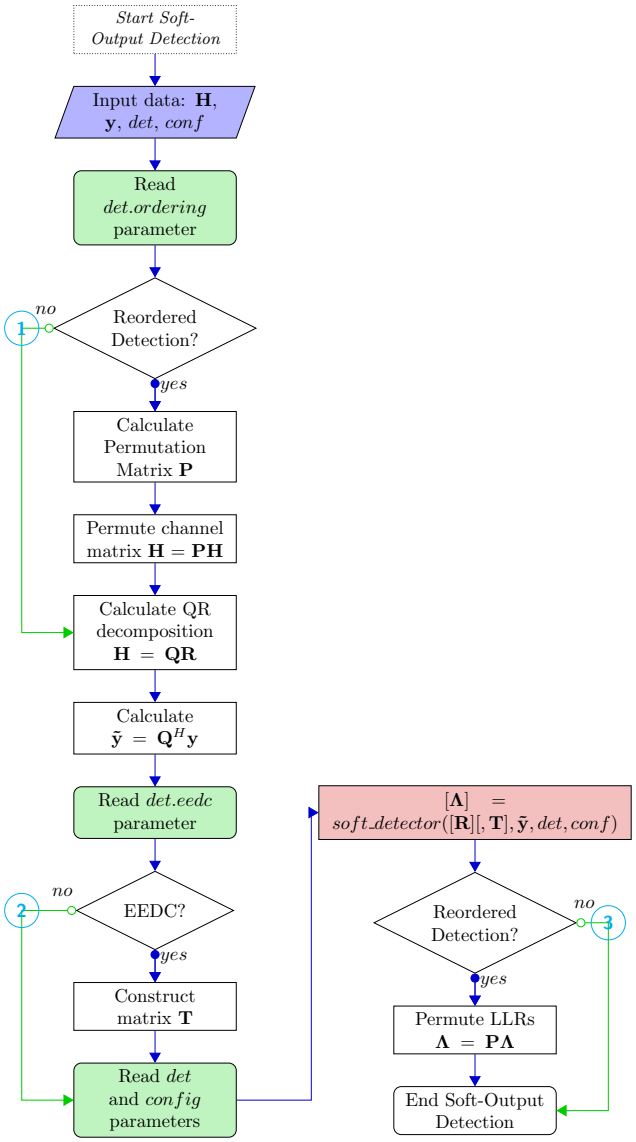


Figure 5.1. Flow Chart of Soft-Output detection.

MIMOPack Soft detector pseudocode

```
function [Λ] = soft_detector(R, ỹ, det, conf)
1.  N_c = size(ỹ, 2)
2.  N_c^gpus = N_c * conf.p_w
3.  if N_c^gpus = 0
4.      omp_set_num_threads(conf.n_cpus)
5.      [Λ_{N_c^gpus+1:N_c,:,:}] = dts_c_name_wrapper(R, ỹ_{:,N_c^gpus+1:N_c}, det)
6.  else if N_c^gpus = N_c
7.      #pragma omp parallel for num_threads(conf.n_gpus)
8.      for dev = 1 : conf.n_gpus
9.          N_dev = N_c^gpus / conf.n_gpus
10.         ini = (dev − 1) · N_dev + 1
11.         end = dev · N_dev
12.         if dev = conf.n_gpus
13.             end = N_dev
14.         end
15.         setDevice(dev)
16.         [Λ_{ini:end,:,:}] = dts_cu_name_wrapper(R, ỹ_{:,ini:end}, det)
17.     end
18. else
19.     omp_set_nested(1)
20.     #pragma omp parallel sections num_threads(2)
21.         #pragma omp section
22.             Get [Λ_{1:N_c^gpus,:,:}] such as steps 7-17
23.         #pragma omp section
24.             Get [Λ_{N_c^gpus+1:N_c,:,:}] such as steps 4-5
25. end
end
```

Figure 5.2. MIMOPack Soft-Output Detector Pseudocode: Calling OpenMP/CUDA wrapper functions Fig. 5.3 and Fig. 5.4.

5.2 Maximum A Posteriori Probability and Max-Log Detectors Implementation

Assuming that all transmit vectors are equally likely, the *optimal soft MAP* (OMAP) *demodulator* calculates the exact LLR for $c_{i,k}$ as

C/OpenMP Soft-Output detection wrapper pseudocode

function $[\Lambda]$ = dts_c_name_wrapper($\mathbf{R}$, $\tilde{\mathbf{y}}$, det)

1. $N_c = size(\tilde{\mathbf{y}}, 2)$
2. **#pragma omp parallel for**
3. **for** $n = 1 : N_c$
4. $\quad [\Lambda_{n,:,:}] = dts\_c\_name(\mathbf{R}, \tilde{\mathbf{y}}_{:,n}[, det.n_E])$
5. **end**

end

Figure 5.3. C/OpenMP Soft-Output detection wrapper pseudocode called from Fig. 5.2. Parameters depend on the detector chosen.

CUDA Soft-Output detection wrapper pseudocode

function $[\Lambda]$ = dts_cu_name_wrapper($\mathbf{R}$, $\tilde{\mathbf{y}}$, det)

1. $[n_T, N_c] = size(\tilde{\mathbf{y}})$
2. $N_s = gpu\_name\_fitting(model, N_c, n_T)$
3. **for** $n = 1 : \frac{N_c}{N_s}$
4. $\quad ini = (n-1) \cdot N_s + 1$
5. $\quad end = n \cdot N_s$
6. $\quad [\Lambda_{ini:end,:,:}] = dt\_cu\_name(\mathbf{R}, \tilde{\mathbf{y}}_{:,ini:end}[, det.n_E][, det.N_{iter}])$
7. **end**

end

Figure 5.4. CUDA Soft-Output detection wrapper pseudocode called from Fig. 5.2. Parameters depend on the detector chosen.

$$\Lambda_{i,k} = log\frac{\mathrm{P}(c_{i,k}=1|\mathbf{y},\mathbf{H})}{\mathrm{P}(c_{i,k}=0|\mathbf{y},\mathbf{H})} = log\frac{\sum_{\mathbf{s}:s_i\in\mathcal{O}_k^1} e^{-\frac{\|\tilde{\mathbf{y}}-\mathbf{R}\mathbf{s}\|^2}{\sigma_w^2}}}{\sum_{\mathbf{s}:s_i\in\mathcal{O}_k^0} e^{-\frac{\|\tilde{\mathbf{y}}-\mathbf{R}\mathbf{s}\|^2}{\sigma_w^2}}}, \tag{5.2}$$

where candidates $\mathbf{s} : s_i \in \mathcal{O}_k^u$ are selected if the bit $[s_i]_k$ is equal to u. Here, $[s_i]$ denotes the binary-valued representation of the constellation symbol s_i. This requires the computation and summation of $|\mathcal{O}|^{n_T}$ probabilities, leading to prohibitive computational complexity. In the decision-tree per-

spective, the MLA needs to compute the euclidean distance of each branch, $\|\tilde{\mathbf{y}} - \mathbf{Rs}\|^2$ in expresion (5.2). Let us consider the detection tree depicted in Fig. 5.5 of a 3×3 MIMO system with QPSK constellation of size $M = 4$ (i.e each constellation symbol is represented with $m = \log_2 M = 2$ bits). As an example, the LLR $\Lambda_{2,1}$ will be calculated using the Euclidean Distances of branches with leafs nodes filled in red (nodes with 1st bit of 2nd symbol set to 1 or $[s_2]_1 = 1$) in the numerator and those filled in blue (nodes with 1st bit of 2nd symbol set to 0 or $[s_2]_1 = 0$) in denominator of equation 5.2.

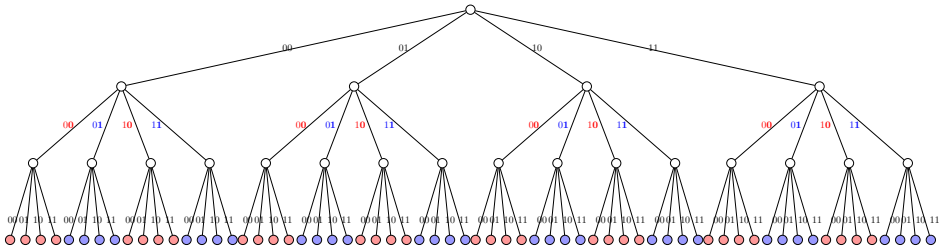


Figure 5.5. Binary-valued representation of the complete decoding tree for a 3×3 using QPSK ($m = 2$) constellation.

Pseudocode described in Fig. 5.6 shows the steps necessary to calculate the LLRs with the *Maximum A Posteriori Probability* (MAP) detector. As can be seen the first part of the algorithm is identical to that described in the MLE detector in section 4.3. Once the Euclidean distances have been calculated for each possible solution $\mathbf{s} \in \mathcal{O}^{n_T}$ using the $fully\_expansion$ algorithm with $n_E = n_T$ (see Chapter 4, Fig. 4.27) the algorithm proceeds calculating the LLRs as Fig. 5.7. $\mathcal{L}$ is a structure which contains, the tree-path formed by n_T constellation symbols in $\mathcal{L}.\mathbf{s}$ and the accumulated Euclidean distance in $\mathcal{L}.\eta$. Note that, the computation of each LLR for each layer i and each bit position k can be done in parallel (step 1 and 3 in Fig. 5.7).

A look-up table (LUT) of $M \times m$ entries, called $\mathcal{G}$, is used in order to find, efficiently, the set of symbols $\mathcal{O}_k^u$ with $k = 1, \ldots, m$ and $u \in \{0, 1\}$. This LUT avoids the need of obtaining each time the representation of s_i in binary format. Matrix $\mathcal{G}$ contains the representation of all constellation symbols in binary format. Table 5.1 shows the LUT $\mathcal{G}$ associated to the detection tree of figure 5.5.

The use of a *Max-Log Approximation* (MLA) detector can reduce the

MAP pseudocode

function $[\mathbf{\Lambda}] = \text{dt\_c\_map}(\mathbf{R}, \tilde{\mathbf{y}}, \sigma_w^2)$
1. $\mathcal{L} = fully\_expansion(\mathbf{R}, \tilde{\mathbf{y}}, n_T)$
2. $\mathbf{\Lambda} = llr\_map(\mathcal{L}, \mathcal{G}, \sigma_w^2)$

end

Figure 5.6. Maximum A Posteriori Probability Detector Pseudocode. The detector calls *fully_expansion* function in Chapter 4, Fig. 4.27 and *llr_map* function in Fig. 5.7.

LLR MAP pseudocode

function $[\mathbf{\Lambda}] = llr\_map(\mathcal{L}, \mathcal{G}, \sigma_w^2)$

1. **#pragma omp parallel for**
2. **for** $i = 1 : n_T$
3. **#pragma omp parallel for**
4. **for** $k = 1 : m$
5. $d^0 = d^1 = 0$
6. **for** $q = 1 : lenght(\mathcal{L})$
7. $x = \mathcal{L}_q.s_i$
8. **if** $\mathcal{G}_{\langle x \rangle, k} = 1$
9. $d^1 = d^1 + e^{-\frac{\mathcal{L}_q \cdot \eta}{\sigma_w^2}}$
10. **else**
11. $d^0 = d^0 + e^{-\frac{\mathcal{L}_q \cdot \eta}{\sigma_w^2}}$
12. **end**
13. **end**
14. $\Lambda_{i,k} = \log(\frac{d^1}{d^0})$
15. **end**
16. **end**

end

Figure 5.7. MAP Calculation of the bits LLRs. This function is called from Fig. 5.6.

complexity of the MAP detector. Algorithm described in Fig. 5.8 shows the pseudocode of the MLA demodulation, where the computation of the LLRs for each code bit is calculated using the max-log approximation as follows (see Fig. 5.9):

Table 5.1. Look-up table $\mathcal{G}$ for a QPSK contellation of size M = 4 and m=2 bits per symbol.

i	k	
	2	1
1	0	0
2	0	1
3	1	0
4	1	1

$$\Lambda_{i,k} \approx \frac{1}{\sigma_w^2}\left[\min_{\mathbf{s}:s_i\in\mathcal{O}_k^0}\|\tilde{\mathbf{y}}-\mathbf{R}\mathbf{s}\|^2 - \min_{\mathbf{s}:s_i\in\mathcal{O}_k^1}\|\tilde{\mathbf{y}}-\mathbf{R}\mathbf{s}\|^2\right]. \tag{5.3}$$

MLA pseudocode

function $[\mathbf{\Lambda}] = \text{dt\_c\_mla}(\mathbf{R}, \tilde{\mathbf{y}}, \sigma_w^2)$
1. $\mathcal{L} = fully\_expansion(\mathbf{R}, \tilde{\mathbf{y}}, n_T)$
2. $\mathbf{\Lambda} = llr\_mla(\mathcal{L}, \mathcal{G}, \sigma_w^2)$

end

Figure 5.8. Max-Log Approximation Detector Pseudocode. The detector calls $fully\_expansion$ function in Chapter 4, Fig. 4.27 and llr_mla function in Fig. 5.9.

5.2.1 CUDA Implementation

The proposed MAP and MLA GPU implementations exhibit a similar algorithmic scheme. Both CUDA codes are composed by two kernels that work together to perform the estimation of the N_s signals.

Algorithm 5.10 shows the needed steps to carry out the detection using GPU. As can be seen, matrix $\mathcal{G}$ contains the representation of all constellation symbols in binary format. This data will not change during the execution and is read only, then can be stored in constant memory in order to get faster memory accesses. The next step launches the $kernel\_mle$ [see

LLR MLA pseudocode

```
function [Λ] = llr_mla(L, G, σ_w^2)
1.  #pragma omp parallel for
2.  for i = 1 : n_T
3.      #pragma omp parallel for
4.      for k = 1 : m
5.          d^0 = d^1 = 1e6
6.          for q = 1 : lenght(L)
7.              x = L_q.s_i
8.              if G_⟨x⟩,k = 1 and L_q.η < d^1
9.                  d^1 = L_q.η
10.             end
11.             if G_⟨x⟩,k = 0 and L_q.η < d^0
12.                 d^0 = L_q.η
13.             end
14.         end
15.         Λ_i,k = (d^0 − d^1) / σ_w^2
16.     end
17. end
end
```

Figure 5.9. MLA Calculation of the bits LLRs. This function is called from Fig. 5.8.

Chapter. 4, Figure 4.16]. Each thread is in charge of computing the accumulated ED for a given signal n and a possible q-th combination of the range $\mathcal{O}^{n_T}$. Note that, the list of pahts $\mathcal{L}$ does not need to be copied to the CPU, since data stored in global memory remains unchanged and can be used for the following kernel.

Once the M^{n_T} Euclidean Distances have been calculated for each signal, the detector must obtain the soft information. Depending on the detector selected (MAP or MLA) Kernel of Fig. 5.11 or Kernel of Fig. 5.12 will be launched. In this case the number of blocks are computed with $n_{th} = N_s \cdot m \cdot n_T$. Each thread is in charge to compute an LLR $\Lambda_{n,i,k}$ for a determinated signal n using Eq. 5.2 or Eq. 5.3 respectively.

5.2.2 Performance Results

In order to assess the performance of the parallel implementations, we have evaluated the execution times of the MLA and MAP detectors described

MAP and MLA Kernel-Launcher Pseudocode

Input: $\mathbf{T} \in \mathbb{C}^{M \times n_v}$, $\tilde{\mathbf{y}} \in \mathbb{C}^{n_R \times N_s}$, σ_w^2
Output: $\mathbf{\Lambda} \in \mathbb{R}^{N_s \times n_T \times m}$

function $[\mathbf{\Lambda}]$ = dt_cu_map($\mathbf{T}$, $\tilde{\mathbf{y}}$, σ_w^2)
1. Allocate and copy $\mathbf{T}$ and $\tilde{\mathbf{y}}$ in GPU GM
2. Allocate output data $\mathcal{L} \in \mathbb{P}^{N_s \times M^{n_T}}$ in GPU GM
3. Select block and grid configuration with $n_{th} = N_s \cdot M^{n_T}$
4. $[\mathcal{L}] = kernel\_mle(\mathbf{T}, \tilde{\mathbf{y}})$
5. Select block and grid configuration with $n_{th} = N_s \cdot n_T \cdot m$
6. Allocate output data $\mathbf{\Lambda}$ in GPU GM
7. Copy look-up table $\mathcal{G}$ in GPU CM
8. **if** MAP detector is selected
9. $\quad [\mathbf{\Lambda}] = kernel\_llr\_map(\mathcal{L}, \sigma_w^2)$
10. **else**
11. $\quad [\mathbf{\Lambda}] = kernel\_llr\_mla(\mathcal{L}, \sigma_w^2)$
12. **end**
13. Copy $\mathbf{\Lambda}$ in CPU

end

Figure 5.10. MAP and MLA Kernel-launcher for N_s time instants. Calling kernels in Fig. 5.11 and 5.12.

in the previous section. The experimental environment used to evaluate the Soft-Ouput detectors is the same as the one mentioned in Chapter 4, Section 4.1.3.

Table 5.2 shows experimental results for MAP and MLA detectors of a 4×4 MIMO system with 16-QAM constellation and $N_c = 10000$ signals.

Comparing costs of the unoptimized versions U with the execution time using a single OpenMP thread, we can see that the EEDC optimization achieves a significant complexity reduction. The detection is done up to 1.20 times faster. On the other hand parallel execution with OpenMP dramatically reduces response time for soft MAP detection running up to 21 times faster than U version. CUDA parallelization outperforms OpenMP implementation for the MAP detector ($\approx 50x$ faster). Nevertheless the speedup with 48 OpenMP for the MLA detector is better than the CUDA one, since the MLA detector has low complexity than MLA algorithm and the LLR calculation needs more comparisons, which implies more warp

Kernel MAP Pseudocode

function $[\mathbf{\Lambda}] = \text{kernel\_llr\_map}(\mathcal{L}, \sigma_w^2)$
1. Calculate using the thread global index z:
2. · Time slot identifier $n \in \{1, ..., N_s\}$
3. · Layer position $i \in \{1, ..., n_T\}$
4. · Bit position $k \in \{1, ..., m\}$
5. **if** $n < N_s$
6. $d^0 = d^1 = 0$
7. **for** $q = 1 : lenght(\mathcal{L})$
8. $x = \mathcal{L}_q.s_i$
9. **if** $\mathcal{G}_{x,k} = 1$
10. $d^1 = d^1 + e^{-\frac{\mathcal{L}_q \cdot \eta}{\sigma_w^2}}$
11. **else**
12. $d^0 = d^0 + e^{-\frac{\mathcal{L}_q \cdot \eta}{\sigma_w^2}}$
13. **end**
14. **end**
15. $\Lambda_{n,i,k} = log(\frac{d^1}{d^0})$
16. **end**

end

Figure 5.11. Computation of the LLR for the MAP by the z-th thread for N_S signals. This kernel is called from Kernel-Launcher of Fig. 5.10.

divergence problems.

5.3 Soft Fixed Sphere Decoder Implementation

In Chapter 4, Section 4.7 the Hard-Output Fixed Complexity SD (FSD) tree search was described for uncoded MIMO detection, which is able to aproximate the performance of the MLE detector combining a channel matrix ordering with a search over a subset of the whole transmit constellation $\mathcal{S} \subset \mathcal{O}^{n_T}$. The FSD detection process can be written as

$$\hat{\mathbf{s}}^{\text{FSD}} = \arg\min_{\mathbf{s}\in\mathcal{S}} \|\mathbf{y} - \mathbf{H}\mathbf{s}\|^2. \tag{5.4}$$

However, the interest in coded transmissions is not only to find the ML solution ($\hat{\mathbf{s}}^{\text{FSD}}$) but also to obtain a set of candidates around the ML solution with different bit values, which can be used to calculate the LLR

Kernel MLA Pseudocode

```
function [Λ] = kernel_llr_mla(ℒ, σ_w^2)
1.  Calculate using the thread global index z:
2.      · Time slot identifier n ∈ {1, ..., N_s}
3.      · Layer position i ∈ {1, ..., n_T}
4.      · Bit position k ∈ {1, ..., m}
5.  if n < N_s
6.      d^0 = d^1 = 1e6
7.      for q = 1 : lenght(ℒ)
8.          x = ℒ_q.s_i
9.          if 𝒢_{x,k} = 1 and ℒ_q.η < d^1
10.             d^1 = ℒ_q.η
11.         end
12.         if 𝒢_{x,k} = 0 and ℒ_q.η < d^0
13.           d^0 = ℒ_q.η
14.         end
14.     end
15.     Λ_{n,i,k} = (d^0 − d^1)/σ_w^2
16. end
end
```

Figure 5.12. Computation of the LLR for the MLA by the z-th thread for N_S signals. This kernel is called from Kernel-Launcher of Fig. 5.10.

information of the transmitted bits $\mathbf{b}$ [see Chapter. 2, Section 2.1.1]. The *Soft Fixed Sphere Demodulation* (SFSD) was proposed in [80] to provide soft information (LLRs) through an improved list of candidates around the ML solution with different bit values.

It is useful to realize that the Euclidean distance of the solution $\hat{\mathbf{s}}^{\text{FSD}}$ of the hard-output FSD detection problem (2.13) directly provides one of the two minima in expression (5.3), denoted in what follows as η^{FSD}:

$$\hat{\mathbf{s}}^{\text{FSD}} = \arg\min_{\mathbf{s}\in\mathcal{S}} \|\mathbf{y} - \mathbf{H}\mathbf{s}\|^2, \qquad \eta^{\text{FSD}} = \left\|\mathbf{y} - \mathbf{H}\hat{\mathbf{s}}^{\text{FSD}}\right\|^2, \tag{5.5}$$

The second minimum in (5.3) for each $i = 1, \ldots, n_T$ and $k = 1, \ldots, m$, can be computed as

Table 5.2. Time Execution comparison in seconds and speedup (S_P) of the MLE detectors for a 4×4 system using 16-QAM constellation with $N_c = 10000$.

		MAP		MLA	
		Time[sec]	S_P	Time[sec]	S_P
	U	305.72	1.00	89.79	1.00
OpenMP Threads	1	290.46	1.18	73.92	1.21
	2	162.65	1.94	39.64	2.27
	4	78.29	3.88	20.81	4.31
	8	42.40	7.28	11.17	8.04
	12	28.76	10.53	7.68	11.69
	24	20.41	15.33	5.98	15.02
	32	17.69	18.38	4.96	18.10
	48	14.45	22.91	3.93	22.85
	GPU	6.15	47.95	5.84	15.38
	(U) GPU	6.76	45.22	5.90	15.22

$$\bar{\eta}_{i,k} = \min_{\mathbf{s}: s_i \epsilon \mathcal{O}_k^{[s_i^{\mathrm{FSD}}]_{\bar{k}}}} \|\mathbf{y} - \mathbf{H}\mathbf{s}\|^2, \tag{5.6}$$

where $[s_i^{\mathrm{FSD}}]_{\bar{k}}$ denotes the bit-negation of bit $[s_i^{\mathrm{FSD}}]_k$. Then, the LLRs can be calculated as show in Fig. 5.16:

$$\Lambda_{i,k} = \frac{1}{\sigma_w^2}(\eta^{\mathrm{FSD}} - \bar{\eta}_{i,k})(1 - 2[\hat{s}_i^{\mathrm{FSD}}]_k). \tag{5.7}$$

The second term in (5.7) is used to adjust the sign depending if η^{FSD} corresponds to the first or the second minimum in (5.3).

The SFSD performs a predeterminated tree-search composed of three different stages (see Fig. 5.14). The first two stages (FE and SE) are known as the hard-output stage or FSD detection:

- A full expansion of the tree (FE) in the first (highest) n_E levels. At the FE stage, for each survivor path, all the possible values of the constellation are assigned to the symbol at the current level.

- A single-path expansion (SE) in the remaining tree-levels $n_T - n_E$. The SE stage starts from each retained path and proceeds down in the tree calculating the solution of the remaining succesive-interference-cancellation (SIC) problem as:

$$\hat{s}_i = \mathcal{Q}\left\{\frac{\tilde{y}_i - \sum_{j=i+1}^{n_T} R_{i,j}\hat{s}_j}{R_{i,i}}\right\}, \quad i = n_T, \ldots, 1. \tag{5.8}$$

- A Soft-Output extension (SOE) to provide soft information by obtaining an improved list of candidates [39]. Figure 5.13 shows the search-tree of the SFSD for the case with $n_T = 3$ and QPSK symbols. The method starts from the list of candidates computed in the FSD detection and adds new candidates to provide more information about the counter bits. Note that, all the possible values of the constellation for each survivor path in the first n_E levels are assigned to the symbol at the current level (i.e. all the necessary values to compute the LLRs of the symbol bits in the first n_E levels are available). Therefore, the list extension must start from the $n_T - n_E$ level of such path.

  To begin the list extension, the best N_{iter} paths are selected from the initial hard-ouput FSD list (in this example, $N_{iter} = 1$). This is based on the heuristics that the lowest-distance paths may be candidates differing from the best paths in only some bits. The symbols belonging to these N_{iter} paths $\hat{s}^{\mathrm{FSD}} = [\mathcal{O}_4\mathcal{O}_2\mathcal{O}_2] = [110101]^T$ are picked up from the root up to a certain level $i = n_T - n_E = 2$, and, at level $i - 1$, additional $\log_2 M$ branches are explored, each of them having one of the bits of the initial path symbol negated (see Fig. 5.15). For example when the first level is expanded, the symbol $[\mathcal{O}_2] = 01$ becomes $0\bar{1} = 00$ negating the 1st bit and $\bar{0}1 = 11$ negating the 2nd bit. In the same way, if the last level is expanded, the symbol $[\mathcal{O}_4] = 11$ becomes $1\bar{1} = 10$ negating the 1st bit and $\bar{1}1 = 01$ negating the 2nd bit. Afterwards, these new partial paths are completed following the SIC path, as done in the hard-ouput FSD scheme. The same operation is repeated until the lowest level of the tree is reached (branches coloured in green in Fig. 5.13).

  In order to accelerate this expansion, a look-up table (LUT) of $M \times m$ entries, called $\mathcal{N}$, is built before the detection. This matrix contains a list of m constellation symbols resulting of the k-th bit negation for

each constellation symbol $\mathcal{O}_i$. For example using the QPSK constellation for the symbol $\mathcal{O}_2$ its binary representation is 01 (see look-up table $\mathcal{G}$ in Table. 5.1). Negating the 1st bit, it becomes $0\bar{1} = 00$, then $\mathcal{N}_{1,1} = 0$, negating the 2nd bit, it becomes $\bar{0}1 = 11$, then $\mathcal{N}_{1,2} = 3$ (see Table 5.3).

Table 5.3. Look-up table $\mathcal{N}$ for a QPSK constellation of size M = 4 and m=2 bits per symbol.

i	k	
	2	1
1	2	1
2	3	0
3	0	3
4	1	2

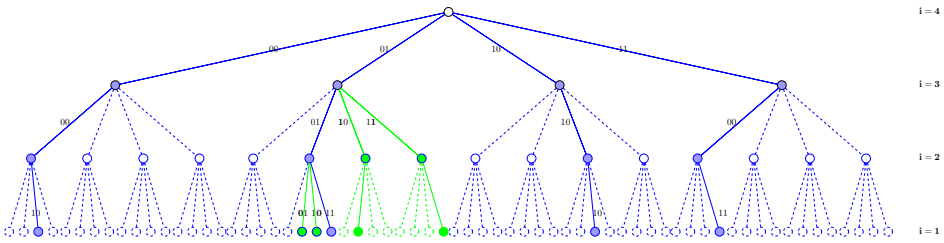


Figure 5.13. List generated by the SFSD algorithm for a 3×3 MIMO system with QPSK modulation ($M = 4$, $m = 2$) and $N_{iter} = 1$.

5.3.1 CUDA Implementation

Algorithm described in Fig. 5.17 shows the steps to perform the SFSD detection. The data for input and output variables are allocated and copied into the GPU-GM memory. In this case, matrices $\mathcal{G}$, $\mathcal{N}$ and constellation symbols $\mathcal{O}$ are copied into constant memory. The $\mathcal{O}$ variable is needed to perform the quantization $\mathcal{Q}(\cdot)$ in the SIC problem. The list of paths $\mathcal{L}$ contains the information of the $M^{n_E} + N_{iter} \cdot m \cdot (n_T - n_E)$ computed paths: M^{n_E} branches of the Hard-Output stage and the $N_{iter} \cdot m \cdot (n_T - n_E)$ new branches of the Soft-Output extension (SOE) stage.

SFSD pseudocode

```
function [Λ] = dt_c_sfsd(R, ỹ, σ_w^2)
1.  [L] = dt_c_hfsd(R, ỹ, n_E) // steps 1-8
2.  [D] = min(L, N_iter)
3.  #pragma omp parallel for
4.  for b = 1 : N_iter
5.      #pragma omp parallel for
6.      for i = n_T − n_E + 1 : −1 : 1
7.          #pragma omp parallel for
8.          for k = 1 : m
9.              path = negated_path(D_b, N, k, i)
10.             L = L ∪ path
11.         end
12.     end
13. end
14. Λ = llr_sfsd(L, G, D_1, σ_w^2)
end
```

Figure 5.14. Soft Fixed Sphere Demodulation Pseudocode. Calling Hard-Output FSD detector function in Chapter 4, Fig. 4.34, list extension algorithm in Fig. 5.15 and llr_sfsd function in Fig. 5.16.

Kernel of Fig. 4.37 used for the FSD detector, is used to calculate the M^{n_E} branches of the FSD stage. At the output, we have the list of candidates paths $\mathcal{L}$. After the hard-output FSD part is finished, the CPU is in charge to calculate the N_{iter} minimum distances and store it in the matrix $\mathcal{D}$ in ascendent order for each signal. This list is copied in the GPU global memory. Then, $kernel\_negated\_paths$ obtains the $N_{iter} \cdot m \cdot (n_T - L)$ new candidates per time index n which are equally distributed among all the threads (see Fig. 5.18). As mentioned, in the SOE stage, additional m branches are explored in the remaining $(n_T - n_E)$ levels. Each of them have one of the bits of the initial path symbol negated. As occurs with matrix $\mathcal{G}$, the matrix $\mathcal{N}$ is constant for the entire simulation, then it will be also copied in constant memory.

After this, the final step finds the minimum distances of paths having

SFSD Negated Path pseudocode

function $[path] = negated\_path(hpath, \mathcal{N}, k, i)$

1. $x = hpath.s_i$
2. $path.\mathbf{s} = hpath.s$
3. $path.\mathbf{s}_i = \mathcal{O}_{\mathcal{N}_{\langle x \rangle, k}}$
4. **for** $l = i - 1 : -1 : 1$
5. $\quad path.s_l = \mathcal{Q}\left\{ \frac{\tilde{y}_l - \sum_{j=l+1}^{n_T} R_{l,j} path.s_j}{R_{l,l}} \right\}$
6. **end**
7. $path.\eta = \sum_{l=n_T}^{1} \left| \tilde{y}_l - \sum_{j=l}^{n_T} R_{l,j} path.s_j \right|^2$

end

Figure 5.15. SFSD Negated Path Pseudocode. This function is called from Fig. 5.14.

the counter bits within the list and computes the $\log_2 M \cdot n_T$ LLRs. These operations are simultaneously executed by the Kernel described in Fig. 5.19.

5.3.2 Performance Results

Due to the lower complexity of the suboptimal SFSD method, we can simulate trasmissions with higher complexity. Figure 5.20 illustrates the speedup comparison using a $n_R \times n_T$ system and $N_c = 10000$ signals, variyng the number of transmitter (n_T) and receiver (n_R) antennas with different constellation sizes. The execution times can be viewed in Tables 5.8, 5.9 and 5.10 for the QPSK, 16-QAM and 64-QAM, respectively. The OpenMP version times have been obtained using 24 OpenMP threads.

The value of N_{iter} is $\{2, 4, 6\}$ for QPSK, 16-QAM and 64-QAM, respectively and $n_E = \lceil \sqrt{n_T} \rceil - 1$. As we can see, the multi-core version have better performance than CUDA version when the computational burden is insufficient to exploit the capabilities of the GPU (i.e. for QPSK constellation). When the number of transmitter antennas n_T and constellation size increases, the CUDA implementation gives better performance than multi-core version. This is more noticeable from $n_T = 10$, since the number of levels in the FE stage is fixed to $n_E = 3$.

LLR SFSD pseudocode

function $[\mathbf{\Lambda}] = llr\_sfsd(\mathcal{L}, \mathcal{G}, minpath, \sigma_w^2)$
1 #**pragma omp parallel for**
2. **for** $i = 1 : n_T$
3. $x = minpath.s_i$
4. #**pragma omp parallel for**
5. **for** $k = 1 : m$
6. $d_{min} = 1e6$
7. **for** $q = 1 : lenght(\mathcal{L})$
8. $xn = \mathcal{L}_q.s_i$
9. **if** $\mathcal{G}_{\langle x \rangle,k} \neq \mathcal{G}_{\langle xn \rangle,k}$ and $\mathcal{L}_q.\eta < d_{min}$
10. $d_{min} = \mathcal{L}_q.\eta$
11. **end**
12. **end**
13. $\Lambda_{i,k} = \frac{(minpath.\eta - d_{min}) \cdot (1 - 2 \cdot \mathcal{G}_{\langle x \rangle,k})}{\sigma_w^2}$
14. **end**
15. **end**
end

Figure 5.16. SFSD Calculation of the bits LLRs. This function is called from Fig. 5.14.

5.4 Fully Parallel Soft Fixed Sphere Demodulation

In SFSD detector, a smart list extension based on the lowest distance paths within the initial FSD list is proposed, however, such extension is performed in an almost totally sequential way, which alters the algorithm parallelism degree. For this reason, we proposed a soft-output demodulator in [75] that performs a fully parallel list extension: the fully parallel FSD (FPFSD). The proposed approach is purely based on the hard-output FSD scheme.

The list of candidates and distances necessary to obtain soft information is calculated through n_T hard-output FSD searches, each with a different channel matrix ordering. The n_T different channel orderings ensure that a different layer (level) of the system is placed at the top of the tree each time. This way, candidate paths containing all the bit labelling possibilities in every level are guaranteed and, thus, soft information about all the bit positions is always available. Recall that, for $n_T = 4$, 4 hard-

SFSD Kernel-Launcher Pseudocode

Input: $\mathbf{T} \in \mathbb{C}^{M \times n_v}$, $\mathbf{T} \in \mathbb{C}^{n_T \times n_T}$, $\tilde{\mathbf{y}} \in \mathbb{C}^{n_R \times N_s}$
Output: $\mathbf{\Lambda} \in \mathbb{R}^{N_s \times n_T \times m}$

function [$\mathbf{\Lambda}$] = dt_cu_sfsd($\mathbf{T}$, $\mathbf{R}$, $\tilde{\mathbf{y}}$)

1. Allocate and copy $\mathbf{T}$, $\mathbf{R}$, $\tilde{\mathbf{y}}$ in GPU GM
2. Allocate and copy output $\mathcal{L} \in \mathbb{P}^{N_s \times M^{n_E} + N_{iter} \cdot m \cdot (n_T - n_E)}$ in GPU GM
3. Select block and grid configuration with $n_{th} = N_s \cdot M^{n_E}$
4. [$\mathcal{L}$] = kernel_hfsd($\mathbf{T}$, $\mathbf{R}$, $\tilde{\mathbf{y}}$)
5. Copy $\mathcal{L}$ in CPU
6. **for** $n = 1 : N_s$
7. $\quad [\mathcal{D}_{n,:}] = min(\mathcal{L}_{n,:}, N_{iter})$
8. **end**
9. Allocate and copy $\mathcal{D}$ in GPU GM
10. Copy $\mathcal{N}$ in GPU CM
11. Select block and grid configuration with $n_{th} = N_s \cdot N_{iter} \cdot m \cdot (n_T - n_E)$
12. [$\mathcal{L}$] = kernel_negated_path($\mathcal{L}$, $\mathcal{D}$)
13. Select block and grid configuration with $n_{th} = N_s \cdot n_T \cdot m$
14. Copy $\mathcal{G}$ in GPU CM
15. [$\mathbf{\Lambda}$] = kernel_llr_sfsd($\mathcal{L}$, $\mathcal{D}$)

end

Figure 5.17. Soft Fixed Sphere Demodulation Kernel-launcher for N_s time instants. Calling kernels in Fig. 4.37, Fig. 5.18 and Fig. 5.19.

output FSD independent searches such as the one in Fig. 4.35 should be carried out, each with a different channel matrix ordering with $n_E = 1$. These tree-searches can be carried out totally in parallel (see Fig. 5.21).

In [39], a special ordering was proposed to place the detection layers associated to the less reliable received symbols at the top of the tree. In this way, FE was performed for those symbols to make the solution independent of the decision in these levels. The rest of the symbols were detected from the most reliable one to the less. This ordering strategy, although showing good performance, involves the calculation of a pseudoinverse matrix with cost $\mathrm{O}(n_T^3)$ at each of the n_T iterations, leading to a complexity of $\mathrm{O}(n_T^4)$. On the other hand, the calculations to be carried out for each iteration are not independent among them, and, no parallelism can be exploited at such preprocessing stage proposed in [39]. Thus, this approach was discarded for our parallel implementation. We propose the use of a much simpler

KERNEL SOE Pseudocode

function $[\mathcal{L}]$ = kernel_negated_paths($\mathcal{L}$, $\mathcal{D}$)
1. Calculate using the thread global index z:
2. · Time slot identifier $n \in \{1, ..., N_S\}$
3. · Path identifier $p \in \{1, ..., N_{iter} \cdot m \cdot (n_T \cdot n_E)\}$
4. · Selected path $b \in \{1, ..., N_{iter}\}$
5. · Layer position $i \in \{1, ..., n_T - n_E\}$
6. · Bit position $k \in \{1, ..., m\}$
7. **if** $n < N_s$
8. $q = M^{n_E} + p$ // Initial position extended list
9. $x = \mathcal{D}_{n,b}.s_i$
11. $\mathcal{L}_{n,q}.\mathbf{s} = \mathcal{D}_{n,b}.\mathbf{s}$
12. $\mathcal{L}_{n,q}.\mathbf{s}_i = \mathcal{N}_{x,k}$
13. **for** $l = i - 1 : -1 : 1$
14. $x = \mathcal{Q}\left\{\frac{\tilde{y}_{l,n} - \sum_{j=l+1}^{n_T} T_{\mathcal{L}_{n,q}.s_j \Gamma_{l,j}}}{R_{l,l}}\right\}$
15. $s_{i,p,n} = \langle x \rangle$
16. **end**
17. $\mathcal{L}_{n,q}.\eta = \sum_{l=n_T}^{1} \left| \tilde{y}_{l,n} - \sum_{j=l}^{n_T} T_{\mathcal{L}_{n,q}.s_j, \Gamma_{l,j}} \right|^2$
18. **end**
end

Figure 5.18. Calculation of new candidates for the SFSD detector by the z-th thread for N_s signals. This kernel is called from Kernel-Launcher of Fig. 5.17.

ordering strategy that is fully parallel and easy to implement in either multi-core or GPU. First, the norms of the columns of the channel matrix are obtained (requiring n_T product, $n_T - 1$ sums, and one squaredroot operation each) and sorted in ascending order (n_T^2 flops in the worst case). Thus, the complexity of this proposed ordering is $O(n_T^2)$. Note that this can be computed considerably faster if the norms are processed in parallel (see Algorithm in Fig. 5.22 step 1). Generally, this ordering leads to more reliable decisions than random ordering since symbols with the highest signal-to-noise ratio are detected before those with the lowest, thus reducing error propagation. Once the norm-based ordering is available in vector **norm**, the n_T orderings needed by the FPFSD are directly built based on

Kernel LLR SFSD Pseudocode

function $[\mathbf{\Lambda}] = \text{kernel\_llr\_sfsd}(\mathcal{L}, \mathcal{D}, \sigma_w^2)$
1. Calculate using the thread global index z:
2. $\quad \cdot$ Time slot identifier $n \in \{1, ..., N_s\}$
3. $\quad \cdot$ Layer position $i \in \{1, ..., n_T\}$
4. $\quad \cdot$ Bit position $k \in \{1, ..., m\}$
5. **if** $n < N_s$
6. $\quad d_{min} = 1e6$
7. $\quad x = \mathcal{D}_{n,1}.s_i$
8. $\quad$ **for** $q = 1 : lenght(\mathcal{L})$
9. $\quad\quad xn = \mathcal{L}_q.s_i$
10. $\quad\quad$ **if** $\mathcal{G}_{x,k} \neq \mathcal{G}_{xn,k}$ and $\mathcal{L}_q.\eta < d_{min}$
11. $\quad\quad\quad d_{min} = \mathcal{L}_q.\eta$
12. $\quad\quad$ **end**
13. $\quad$ **end**
14. $\quad \Lambda_{n,i,k} = \frac{(\mathcal{D}_{n,1}.\eta - d_{min}) \cdot (1 - 2 \cdot \mathcal{G}_{x,k})}{\sigma_w^2}$
15. **end**

end

Figure 5.19. Computation of the LLR for the SFSD by the z-th thread for N_s signals. This kernel is called from Kernel-Launcher of Fig. 5.17.

this initial norm-based ordering. Taking into account the aforementioned requirements, the first level of each new ordering is assigned a different detection position to guarantee the availability of soft information for all the possible bit values in every system level.

Note that, as when using the FSD ordering, the reliability of the symbol placed in the FE stage is irrelevant. Then, the remaining levels are ordered following the initial column-norm-based ordering but skipping the level that was already set on the top ($\Delta = \{norm_i\} \cup (\{1, 2, \cdots, n_T\} - \{norm_i\})$). The example in Table 5.7 shows how the ordering is set up for a particular column-norm-based ordering of a 4×4 channel, which in this case is $\{2, 4, 3, 1\}$. As the first row of Table 5.7 shows, the ith proposed ordering starts the data detection at the i-th tree-level, being $i \in \{1, 2, 3, 4\}$. Then, the remaining levels are explored following the column-norm-based

Table 5.4. Time Execution comparison in seconds of SFSD detector with different library configurations for a system using QPSK constellation as a function of the number of transmitted antennas (n_T).

n_T	1 OpenMP thread (U)	24 OpenMP threads	GPU
2	$0.2 \cdot 10^{-1}$	$0.06 \cdot 10^{-1}$	$3.11 \cdot 10^{-1}$
4	$0.7 \cdot 10^{-1}$	$0.10 \cdot 10^{-1}$	$3.04 \cdot 10^{-1}$
8	$2.52 \cdot 10^{-1}$	$0.30 \cdot 10^{-1}$	$3.14 \cdot 10^{-1}$
10	$7.72 \cdot 10^{-1}$	$0.45 \cdot 10^{-1}$	$3.59 \cdot 10^{-1}$
12	$9.73 \cdot 10^{-1}$	$0.56 \cdot 10^{-1}$	$3.50 \cdot 10^{-1}$
14	$13.10 \cdot 10^{-1}$	$0.73 \cdot 10^{-1}$	$3.34 \cdot 10^{-1}$
16	$16.50 \cdot 10^{-1}$	$0.92 \cdot 10^{-1}$	$3.67 \cdot 10^{-1}$
18	$21.66 \cdot 10^{-1}$	$1.14 \cdot 10^{-1}$	$3.85 \cdot 10^{-1}$
20	$26.21 \cdot 10^{-1}$	$1.37 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$

Table 5.5. Time Execution comparison in seconds of SFSD detector with different library configurations for a system using 16-QAM constellation as a function of the number of transmitted antennas (n_T).

n_T	1 OpenMP thread (U)	24 OpenMP threads	GPU
2	0.27	0.11	0.35
4	0.80	0.12	0.36
8	2.21	0.15	0.37
10	33.48	1.88	1.07
12	44.08	2.39	1.22
14	57.60	3.05	1.42
16	71.11	3.60	1.70
18	89.96	4.43	2.00
20	103.59	5.40	2.41

ordering in column 2.

Table 5.6. Time Execution comparison in seconds of SFSD detector with different library configurations for a system using 64-QAM constellation with $N_c = 10000$ as a function of the number of transmitted antennas (n_T).

n_T	1 OpenMP thread (U)	24 OpenMP threads	GPU
2	3.35	0.42	0.77
4	12.03	0.70	0.92
8	33.42	1.86	1.27
10	2812.85	138.71	70.51
12	3719.92	186.10	81.56
14	4590.03	233.80	96.49
16	5586.36	293.54	114.71
18	6604.37	358.22	131.94
20	7691.20	444.28	151.81

Table 5.7. Symbol detection position and corresponding tree-level for the involved FPFSD orderings in an example with $n_T = 4$

Detection position	Norm-based Ordering	Order 1	Order 2	Order 3	Order 4
1st	2	1	2	3	4
2nd	4	2	4	2	2
3rd	3	4	3	4	3
4th	1	3	1	1	1

5.4.1 CUDA Implementation

Algorithm 5.23 shows the steps needed to perform the FPFSD detection. Once the relevant data has been allocated and copied in the GPU. Kernel in Fig. 5.24 calculates the EDs of the M branches for each f-th order matrix and n time instants. Once the Euclidean Distances have been calculated, the detector must obtain the soft information. The LLRs are computed using Kernel in Fig. 5.12.

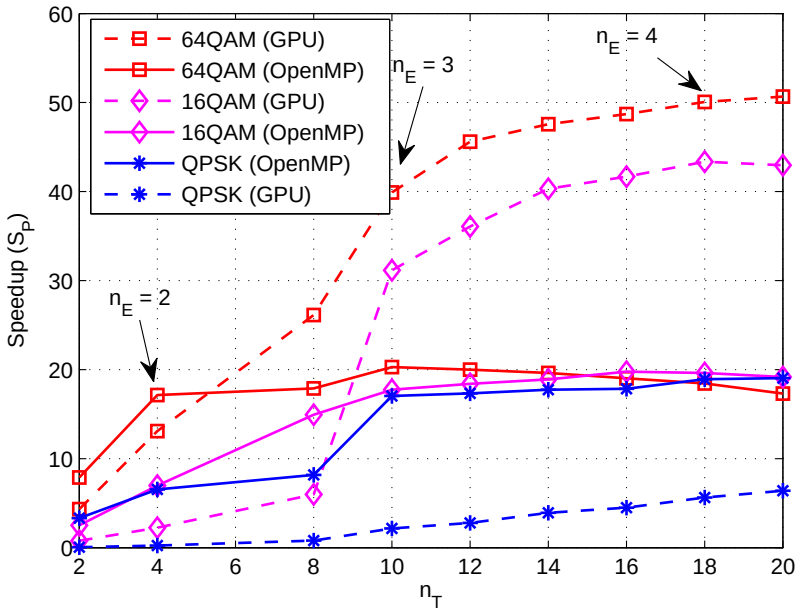


Figure 5.20. Speedup (S_P) comparison for the SFSD detector for a $n_R \times n_T$ system considering $N_c = 10000$ with different constellations and number of transmitter antennas.

5.4.2 Performance Results

As in the case of SFSD scheme, the lower complexity of the suboptimal FPFSD method allows to simulate trasmissions with higher complexity. Figure 5.25 has been obtained using a $n_R \times n_T$ system considering $N_c = 10000$ signals variyng the number of transmitter and receiver antennas (n_T) and constellation types. The execution times can be viewed in Tables 5.8, 5.9 and 5.10 for the QPSK, 16-QAM and 64-QAM, respectively. The OpenMP version times have been obtained using 24 OpenMP threads.

The speedup results of the parallel FPFSD implementations can be seen in Figure 5.25. As we can be observed, the multi-core version shows better performance than CUDA version when the size of the constellation is small. When the number of transmitter antennas n_T and constellation size increases, the CUDA implementation exhibits better performance than the multi-core version.

FPFSD pseudocode

function $[\mathbf{\Lambda}] = \text{dt\_c\_fpfsd}(\mathbf{R}, \tilde{\mathbf{y}})$
1. $\mathcal{L} = \emptyset$
2. **#pragma omp parallel for**
3. **for** $i = 1 : n_T$
4. $[\mathcal{D}] = dt\_c\_hfsd(\mathbf{R}_{:,:,i}, \tilde{\mathbf{y}}_{:,i}, 1)$
5. $\mathcal{L} = \mathcal{L} \cup \mathcal{D}$
6. **end**
7. $\mathbf{\Lambda} = llr\_mla(\mathcal{L}, \mathcal{G}, \sigma_w^2)$

end

Figure 5.21. Fully Parallel Soft Fixed Sphere pseudocode. Calling Hard-Output FSD detector function in Chapter 4, Fig. 4.34 and *llr_mla* function in Fig. 5.9.

FPFSD PREPROCESSING PSEUDOCODE

INPUT: $\mathbf{H} \in \mathbb{C}^{n_R \times n_T}$, $\mathbf{y} \in \mathbb{C}^{n_R \times N_c}$
OUTPUT: $\mathbf{R} \in \mathbb{R}^{n_T \times n_T \times n_T}$, $\tilde{\mathbf{y}} \in \mathbb{R}^{n_R \times n_T \times N_c}$

function $[\mathbf{R}, \tilde{\mathbf{y}}] = \text{fpfsd\_precoding}(\mathbf{H}, \mathbf{y})$
1. **#pragma omp parallel for**
2. **for** $i = 1 : n_T$
3. $norm_i = \|\mathbf{H}_{:,i}\|^2$
4. **end**
5. $[\mathbf{norm}] = sort(\mathbf{norm})$
6. **#pragma omp parallel for**
7. **for** $i = 1 : n_T$
8. $\Delta = \{norm_i\} \cup (\{1, 2, \cdots, n_T\} - \{norm_i\})$
9. $\mathbf{H}_{:,\Delta} = \mathbf{QR}_{:,:,i}$
10. **#pragma omp parallel for**
11. **for** $n = 1 : N_c$
12. $\tilde{\mathbf{y}}_{:,i,n} = \mathbf{Q}^H \mathbf{y}_{:,n}$
13. **end**
14. **end**

end

Figure 5.22. Fully Parallel FSD preprocessing stage pseudocode.

FPFSD Kernel-Launcher Pseudocode

Input: $\mathbf{T} \in \mathbb{C}^{M \times n_v}$, $\mathbf{R} \in \mathbb{C}^{n_T \times n_T \times n_T}$, $\tilde{\mathbf{y}} \in \mathbb{C}^{n_T \times n_T \times N_s}$
Output: $\mathbf{\Lambda} \in \mathbb{R}^{N_s \times n_T \times m}$

function $[\mathbf{\Lambda}] =$ dt_cu_hfsd($\mathbf{T}$, $\tilde{\mathbf{y}}$)
1. Allocate and copy $\mathbf{T}$, $\mathbf{R}$ and $\tilde{\mathbf{y}}$ in GPU GM
2. Allocate output data $\mathcal{L} \in \mathbb{P}^{N_s \times n_T \cdot M}$ in GPU GM
3. Copy constellation symbols $\mathcal{O}$ in GPU CM
4. Select block and grid configuration with $n_{th} = N_s \cdot n_T \cdot M$
5. $[\mathcal{L}] =$ kernel_fpfsd($\mathbf{T}$, $\mathbf{R}$, $\tilde{\mathbf{y}}$)
6. Select block and grid configuration with $n_{th} = N_s \cdot m \cdot n_T$
7. $[\mathbf{\Lambda}] = kernel\_llr\_mla(\mathcal{L}, \sigma_w^2)$
8. Copy $\mathbf{\Lambda}$ in CPU

end

Figure 5.23. Fully Parallel FSD Kernel-launcher for N_s time instants. Calling algorithm in Fig. 5.24 and kernel in Fig. 5.12.

KERNEL FPFSD Pseudocode

function $[\mathcal{L}] =$ kernel_fpfsd($\mathbf{R}$, $\tilde{\mathbf{y}}$)
1. Calculate using the thread global index z:
2. $\quad \cdot$ Time slot identifier $n \in \{1, ..., N_s\}$
3. $\quad \cdot$ Tree-Path identifier $p \in \{1, ..., M\}$
4. $\quad \cdot$ FPFSD ordering index $f \in \{1, ..., n_T\}$
5. **if** $n < N_s$
6. $\quad q = f \cdot M + p$
7. $\quad \mathcal{L}_{n,q}.\mathbf{s}_{n_T} = q$
8. $\quad$ **for** $i = n_T - n_E : -1 : 1$
9. $\quad\quad x = \mathcal{Q}\left\{ \frac{\tilde{y}_{i,f,n} - \sum_{j=i+1}^{n_T} R_{i,j,f} \mathcal{L}_{n,q}.s_j}{R_{i,i,f}} \right\}$
10. $\quad\quad \mathcal{L}_{n,q}.s_i = \langle x \rangle$
11. $\quad$ **end**
12. $\quad \mathcal{L}_{n,q}.\eta = \sum_{i=n_T}^{1} \left| \tilde{y}_{i,f,n} - \sum_{j=i}^{n_T} R_{i,j,f} \mathcal{O}_{\mathcal{L}_{n,q}.s_j} \right|^2$
13. **end**

end

Figure 5.24. Fully Parallel FSD Kernel pseudocode. This kernel is called from Fig. 5.23.

Table 5.8. Time Execution comparison in seconds of FPFSD detector with different library configurations for a system using QPSK constellation as a function of the number of transmitted antennas (n_T).

n_T	1 OpenMP thread (U)	24 OpenMP threads	GPU
2	$0.14 \cdot 10^{-1}$	$0.13 \cdot 10^{-1}$	$3.105 \cdot 10^{-1}$
4	$0.66 \cdot 10^{-1}$	$0.10 \cdot 10^{-1}$	$3.03 \cdot 10^{-1}$
8	$2.85 \cdot 10^{-1}$	$0.31 \cdot 10^{-1}$	$3.23 \cdot 10^{-1}$
10	$4.34 \cdot 10^{-1}$	$0.73 \cdot 10^{-1}$	$3.22 \cdot 10^{-1}$
12	$6.24 \cdot 10^{-1}$	$1.20 \cdot 10^{-1}$	$3.48 \cdot 10^{-1}$
14	$8.39 \cdot 10^{-1}$	$1.16 \cdot 10^{-1}$	$3.36 \cdot 10^{-1}$
16	$11.31 \cdot 10^{-1}$	$2.42 \cdot 10^{-1}$	$3.34 \cdot 10^{-1}$
18	$15.54 \cdot 10^{-1}$	$2.58 \cdot 10^{-1}$	$3.37 \cdot 10^{-1}$
20	$19.58 \cdot 10^{-1}$	$2.97 \cdot 10^{-1}$	$4.13 \cdot 10^{-1}$

Table 5.9. Time Execution comparison in seconds of FPFSD detector with different library configurations for a system using 16-QAM constellation as a function of the number of transmitted antennas (n_T).

n_T	1 OpenMP thread (U)	24 OpenMP threads	GPU
2	0.06	0.01	0.31
4	0.23	0.03	0.32
8	1.11	0.22	0.32
10	1.79	0.32	0.35
12	2.58	0.40	0.37
14	3.69	0.34	0.38
16	5.03	0.37	0.42
18	6.73	0.44	0.45
20	8.70	0.53	0.52

5.5 Conclusions

This chapter has presented a set of Soft-Output detectors. The MAP and MLA detectors are the best in terms of bit error rate but require high computing resources. For this reason two detectors based on the fixed-

Table 5.10. Time Execution comparison in seconds of FPFSD detector with different library configurations for a system using 64-QAM constellation with $N_c = 10000$ as a function of the number of transmitted antennas (n_T).

n_T	1 OpenMP thread (U)	24 OpenMP threads	GPU
2	0.28	0.03	0.31
4	1.11	0.18	0.33
8	4.98	0.40	0.42
10	8.39	0.57	0.53
12	12.59	0.76	0.63
14	18.00	1.04	0.78
16	24.58	1.38	0.92
18	32.37	1.79	1.15
20	41.44	2.26	1.36

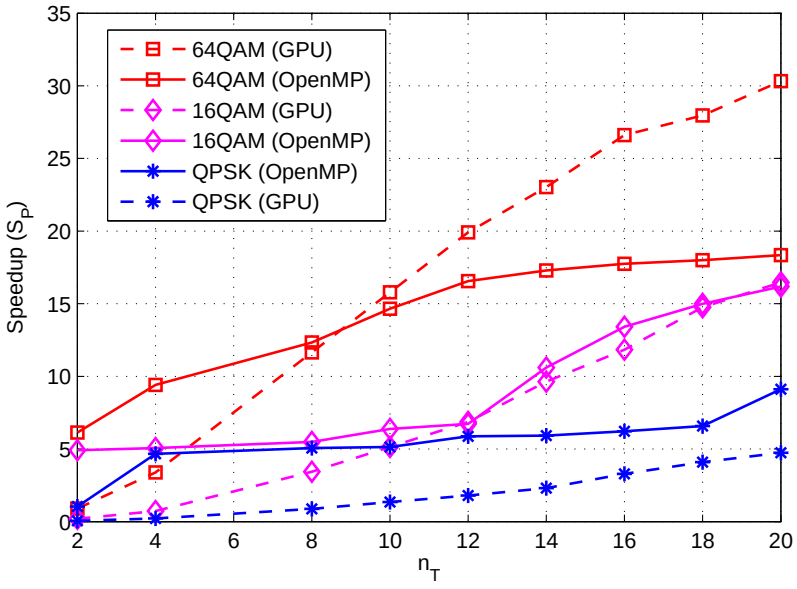


Figure 5.25. Speedup (S_P) comparison for the FPFSD detector for a $n_R \times n_T$ system considering $N_c = 10000$ with different constellations and number of transmitter antennas.

complexity sphere decoder have been implemented in MIMOPack library.

A study of the execution time for different number of signals, system sizes and constellation sizes was carried out to evaluate the influence of these parameters on the library performance.

The selected methods were also implemented in a sequential and unoptimized version. The execution times of the both multi-core and GPU parallel soft-output schemes were compared to the execution times of the sequential and unoptimized versions. Speedup comparison showed that the multi-core and GPU-based detector performs up to 30 times faster than sequential versions for some cases. For the MAP and SFSD detector the GPU detects 50 times faster than the unoptimized version. The parallell implementations is shown to be scalable when the constellation and MIMO system sizes are large.

In the previous chapter, the impact of the EEDC optimization on the Hard-Output detector performance was determined. In this chapter it has been confirmed that indeed the use of this technique boost also the efficiency of the Soft-Output detection up to 20%.

As in the Hard-Output detection, future work is needed to optimize the detection stage using the features of the different GPUs.

Finally, it can be remarked that some information reported in this chapter has been published in international journals. The OpenMP implementation of the SFSD has been reported in [76]. The SFSD was used for the implementation of a iterative receiver for energy saving MIMO ID-BICM systems in [81]. The paper containing the design and GPU implementation using shared memory and different GPU features of the FPFSD is reported in [75] and [82].

6 MIMOPack Software Package

6 MIMOPack Software Package

This chapter presents MIMOPack, a set of optimized functions to perform some of the most complex stages in Multiple-Input Multiple-Output (MIMO) communication systems. It is a meaningful software package that implements a set of Hard and Soft output detectors. These detectors are highly configurable and have been optimized to run on a wide range of architectures increasing the portability of scientific codes between different computing environments. MIMOPack aims to become a usefull library for the research community facilitating the development of versatile and scalable parallel applications and also to speed up simulation platforms, which are commonly used to assess different technologies proposed by companies involved in standardization processes. MIMOPack is freely-available and it can be downloaded from `http://www.inco2.upv.es/mimopack/index.html`.

6.1 Introduction and Objectives

The existence of libraries becomes a useful tool that allows the specialist in a particular field to focus on solving their specific problems and save hours of programming some optimized computing routines. Nowadays, there exist

different signal processing libraries covering a wide range of scientific and technological applications as Aquila [83], Signal Processing Toolbox [84], Communication System Toolbox [18] and IT++ [19]. Unfortunately few of them offer specific functions for simulating MIMO communication systems able to take advantage of high performance computers.

For example, Communication System Toolbox provide different functions to take profit of parallel computing over multi-core processors, but most of them are not prepared to work on GPUs. Similarly, IT++ provides mathematical, signal processing and communication functions but nowadays it does not have any kind of support to use GPUs. There is an unfilled space in the field of communication systems libraries: a multi-core/GPU-oriented library for MIMO communication sytems. It is true that it involves a broad set of processing methods that are dependent on the channel conditions and systems parameters. The creation of a library with these features is a challenge both of scientific and technological importance.

The library proposed in this document, named MIMOPack [85], aims at solving some of the most complex stages in a MIMO communication system. These problems are fundamentally: preprocessing techniques, hard and soft output detectors and precoding techniques. Our main goal is to provide with a high performance library that will help to ease the implementation of codes without having to know different programming languages and machine architectures. The design of this library must cope with the following features:

1. High performance computing. It should allow optimum performance on today's computers with parallel architectures and easy adaptation to new implementations as they come.

2. Portability. It should be possible its use on different hardware/software environments, without fundamental changes of the code.

3. Friendly. It should provide the possibility of its general use even for non-expert users.

4. Gradual development. Software and hardware evolution must be had in mind. This will also allow the feedback needed to ensure the quality of developments.

This chapter presents the basic ideas that have helped to design the library and give an overview of its functionality. The current form of the library and what are the next updates to be included in the short term are also described.

6.2 Design and Specifications

The library is written in C language and composed by several modules. Double-precision complex version is available. The library is continuosly growing, the updated release collects a set of functions divided in blocks depicted in Figure 6.1:

- Hard-Output detection:
  - Linear: Zero Forcing with Succesive Interference Cancellation.
  - Maximum Likelihood: ML Exhaustive Decoder, Schnorr Euchner Sphere Decoder and Automatic Sphere Decoder.
  - Fixed Complexity: K-Best Sphere Decoder and Hard Fixed Complexity Sphere Decoder.
  - Wintress framework: it is a divide and conquer framework devised to ease the parallelization of any kind of hard output detector but it can be used also for sequential detector implementations.
- Soft-Output detection:
  - Maximum Likelihood: Maximum A Posteriori Probability and Max-Log detectors.
  - Fixed Complexity: Soft Fixed Complexity Sphere Decoder and Fully Parallel Fixed Complexity Sphere Decoder.
- Preprocessing:
  - Channel matrix ordering: Column-Norm-Based, Fixed Sphere Decoder and Fully Parallel FSD orderings.
  - Channel matrix decomposition: QR decomposition computed with a series of Givens rotations.

- Modulation/Demodulation: functions prepared for BPSK, QPSK, 16QAM and 64QAM constellations.
- Channel: Random AWGN channel and channel emulator (i.e. emulate the transmission go through a matrix channel).
- Detection Tree functions: K-Best and radius tree-pruning, Euclidean Distance Calculation, Efficient Euclidean Distance Calculation, etc.
- Simulation: random simulation data and various performance functions (e.g. BER, SER, execution time, throughput, etc.).
- Configuration: different functions to set and check the platform configuration.
- Mathematical: includes mathematical functions for complex numbers.
- Miscellaneous: printing, sorting and miscellaneous functions.

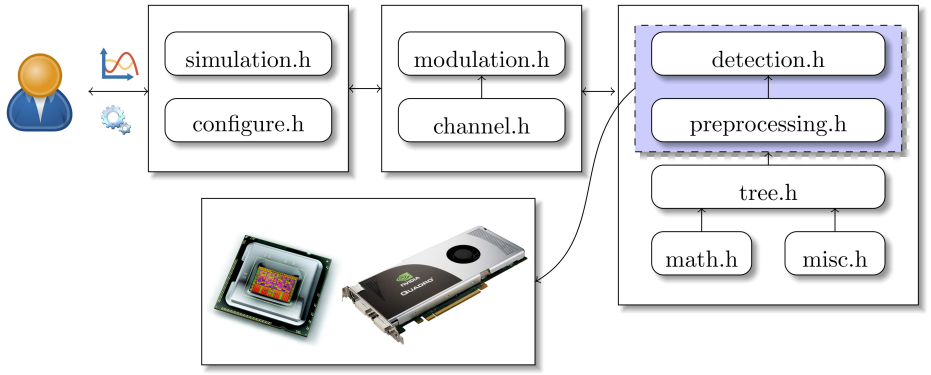


Figure 6.1. Simulation chain through the MIMOPack library modules.

The interface of the functions are common to all environments in order to ease its use, regardless of the machine where it was executed. This feature increases the portability of codes between different computing platforms. Nowadays, it supports the execution of the routines on a sequential/multi-core processor, GPU/multi-GPU devices and heterogeneous mode (i.e. multi-core/multi-GPU). Only NVIDIA GPUs are supported. It is expected in

future releases to deal with AMD GPUs using OpenCL and Intel Xeon Phi coprocessor. Furhtermore, most of them ready to be called from MATLAB through MEX-Files.

6.3 Documentation and Website description

The website structure of MIMOPack is organized in a usual manner (see Figure 6.2). Thus, it has a "Main page" and some tabs for "Installation", "Documentation", "Test", "FAQs" and so on. The main page shows a presentation of the website, the licensing and last version news (see Fig. 6.2). Besides, a brief description of the research group involved in the development of the package and the related projects are outlined. The most relevant tabs provide information like:

- Documentation. A link to the Doxygen generated documentation is shown. In this documentation we can find all the library API specifications.

- Test. In this tab it is shown how to carry out performance tests to obtain the timing and precision results of the installed package in the machine.

- FAQs. The "Frequently Asked Questions" tab includes general questions, installation questions, how to use or program with MIMOPack, questions or problems about different platforms or operating systems and miscellaneous questions.

- References. Includes bibliographic references used to produce MIMOPack: list of published work produced during the MIMOPack design and other related publications.

- Installation. It contains two sections: "How to install" and "Download". In the "Download" section the last version of the software can be chosen and downloaded.

Figure 6.2. MIMOPack Website: Home

6.4 Support and Development

MIMOPack emerged as a result of various research projects in the Interdisciplinary Computation and Communication Group (INCO2). MIMOPack has provided several publications and research works developed for many years. See for example references [75][76][79][81][85]. Among the projects that have contributed to the development of MIMOPack only those projects actives currently are citated:

- PROMETEO II: High Performance Computation and Communications and Applications in Engineering (PROMETEOII/2014/003. Generalitat Valenciana. Spain.)

- DISCOSOUND: Distributed and Collaborative Processing of Sound Signals: Algorithms, Tools and Applications (TEC2012-38142-CO4).
- PROMETEO: High Performance Computing Tools for solving Signal Processing Problems on Parallel Architectures (PROMETEO/2009/013. Generalitat Valenciana. Spain.)
- SMaGGic: Spatial audio systems based on Massive parallel processing of multichannel acoustic signals with General Purpose Graphics Processing Units (GPGPU) and Multi-cores (TEC2009-13741. Ministerio de Ciencia e Innovación. Spain).
- COPABIB: Automatic Development and Tuning of Parallel Libraries for Scientific Computation (TIN2008-06570-C04-02. Ministerio de Ciencia e Innovación. Spain).
- PAID-UPV: Development and Implementation of Computational Kernels for Software Defined Radio Platforms on Multicore/Manycore Architectures (PAID-05-10).

Although initially MIMOPack is the work of the mentioned group, author intention is to incorporate to MIMOPack developments of others researchers that meet the specifications, objectives and philosophy that surround the library. In this sense, through the library's Web site it is possible to contact the author both for discuss ideas in relation to the codes or for collaborating on the development of new routines.

6.5 Configurability and Data Structures

In this section data structures used by the library to configure each of the stages or elements of a simulation are presented. It also shows some of the functions that perform the initialization of these variables. The modules that can be configured are: execution platform, modulation, detector, WinTrees framework and MIMO system simulation.

6.5.1 Platform Configuration

The library allows to assign the type and number of resources to use during the execution. For a given C source code, the user may specify a different

distribution of the computational resources for each function through the configuration of the **mmp_config** structure, which comprises the following parameters:

```
typedef struct{
    int n_cpus:
    int n_gpus:
    double pw:
}mmp_config;
```

where:

- int n_cpus: Number of OpenMP threads.
- int n_gpus: Number of GPUs. The library automatically selects those with more CUDA cores on the detected devices.
- double pw: Percentage of workload that will be delivered among the n_gpus. The remaining workload (1 - pw) will be computed by n_cpus OpenMP threads.

Two commands are currently provided to allow the correct configuration of the environment: **set_mmp_config** and **check_config**. The first function is used to fix parameters of the *mmp_config* structure. In order to ensure the proper configuration, the library implements the function **check_config**, which checks the properties of the execution platform and indicates the setup incompatibilities, if any. These routines can be called from a C source code as follows:

```
mmp_config set_mmp_config(int n_gpus, int n_cpus, double pw);
void check_config(mmp_config config);
```

A calling example could be the following. We are going to use a configuration with 80% of workload for two GPUs and 20% of workload for 12 OpenMP threads:

```
mmp_config my_configuration;
my_configuration = set_mmp_config(2, 12, 80);
check_config(my_configuration);
```

6.5.2 QPSK and QAM Modulation Configuration

The modulation in MIMOPack is defined by using **mmp_modulator** structure. The real and imaginary parts of the complex variables (e.g constellation, channel matrix, transmitted signals, etc.) are stored independently: real (denoted with the prefix "_r") and imaginary (denoted with the prefix "_i"):

```
typedef struct{
    int M;
    int m;
    double *cons_r;
    double *cons_i;
    double *PM;
    double *bounds;
    long int *pattern;
    int *gray_lut;
    int *neg_lut;
}mmp_modulator;
```

where:

- int M: Modulation or constellation size $M \in \{2, 4, 16, 64\}$.
- int m: Number of bits to represent each constellation point ($m = log_2 M$).
- double *cons_r: Real part of the constellation values.
- double *cons_i: Imaginary part of the constellation values.
- double *PM: Real-valued representation of constellation.

- double *bounds: Bounds to quantize when the constellation is expressed as a set of consecutive integers.
- long int *ppattern: It contains the pruning pattern vector $\mathcal{P}$ used for the SESD detector.
- int *gray_lut: Look-up table $\mathcal{G}$. It contains the representation of all constellation symbols in binary format $\mathcal{G}$. This variable is used in the LLR computation by the Soft-Output detectors.
- int *neg_lut: Look-up table $\mathcal{N}$. It contains, a list of m constellation symbols resulting of the k-th bit negation for each constellation symbol, $\mathcal{O}_i$. This variable is used in the LLR computation by the SFSD detector.

The following function can be used to construct the modulator. The user provides only the size of the constellation and the number of transmitter antennas, the library itself fills the rest of the fields when the function is executed:

mmp_modulator **set_mmp_modulator**(int M, int nT);

6.5.3 MIMOPack Detector Configuration

Each detector can be initialized filling the following **mmp_detector** structure in order to select not only the type of the detector but also different optimization techniques to be used in the detection module. This structure is composed by the following fields:

```
typedef struct{
    char *name;
    char *ordering;
    int nE;
    int K;
    double *r;
    double n_v;
    bool eedc;
}mmp_detector;
```

where:

- char *name: Detector type, being name ∈ {"zfsic", "mle", "sdse", "asd", "kbest", "hfsd", "map", "mla", "sfsd", "fpfsd"}.
- char *ordering: Matrix ordering type, being ordering ∈ {"cbo", "hfsd", "fpfsd", "none"}.
- int nE: Number of fully expansion levels.
- int K: Number of survivors for K-best detector.
- int Niter: Selected paths in the SFSD-SOE stage.
- double *r: Initial radius for sphere decoders. This array must contain a radius for each signal considered in the simulation.
- bool eedc: It indicates if Efficient Euclidean Distances optimization must be used.

Different functions are provided by the package to configure each detector:

```
mmp_detector
  set_dt_zfsic(char *ordering, bool eedc);
  set_dt_mle(char *ordering, bool eedc);
  set_dt_sesd(char *ordering, bool eedc, double *r, int Nc);
  set_dt_asd(char *ordering, bool eedc, double *r, int Nc);
  set_dt_kbest(char *ordering, bool eedc, int K, int nE);
  set_dt_hfsd(char *ordering, bool eedc, int nE);
  set_dt_map(char *ordering, bool eedc);
  set_dt_mla(char *ordering, bool eedc);
  set_dt_sfsd(char *ordering, bool eedc, int nE, int Niter);
  set_dt_fpfsd(char *ordering);
```

A calling example is the following. We are using a Sphere Decoder using a common initial radius = ∞ to decode $N_c = 1000$ signals with column-based-norm ordering and EEDC optimization:

```
mmp_detector my_detector;
int i;
int Nc=1000
double *radius = (double *) malloc(Nc*sizeof(double));

for(i = 0; i < Nc; i++){
   radius[i] = 1e6;
}
my_detector = set_dt_sesd("cbo", true, radius, Nc);
```

6.5.4 MIMOPack WinTrees Framework Configuration

If WinTrees framework is used, the **mmp_framework** structure should be defined, where is selected: the type of pruning, the number of levels to be fully expanded and the MIMOPack detector. This structure has the following fields:

```
typedef struct{
    char *prune;
    char *ordering;
    int nE;
    int nK;
    double r;
    bool use_mmp;
    mmp_detector detector;
}mmp_framework;
```

where

- char *prune: Prunning type, being prune $\in$ {"kbest", "radius", "none"}.
- char *ordering: Matrix ordering type, being ordering $\in$ {"cbo", "hfsd", "none"}.

- int nE: Number of fully expansion levels.
- int nK: Number of survivors for K-best considered in the pruning stage.
- double r: Radius considered in the pruning stage.
- bool use_mmp: It indicates if MIMOPack detector must be used in each subtree.
- mmp_detector detector: MIMOPack detector configuration.

The following function can be used to initialise the framework:

mmp_framework **set_framework**(int nE, int nK, double r, char *prune, char *ordering, mmp_detector detector, bool use_mmp);

6.5.5 MIMO Simulation Configuration

The data structure that works with the simulation contains the whole information related to the MIMO system that is currently being simulated. Precise data of the simulation such as the number of signals to be transmitted (Nc) or variation of the channel (Lch) is also considered. Furthermore, this data structure stores the performance results after the detection, such as: bit error rate, symbol error rate and throughput.

```
typedef struct{
    int nT,
    int nR,
    double snr,
    double pot_s,
    int Nc,
    int Lch,
    double ber;
    double ser;
    double bps;
    double itime;
    double etime;
    mmp_dt_time dt_time;
}mmp_simulation;
```

where:

- int nT: Number of transmitter antennas.
- int nR: Number of receiver antennas.
- double snr: Signal-to-Noise Ratio in dB.
- double pot_s: Power of the transmitted vector.
- int Nc: Number of signals to be transmitted.
- int Lch: Variation of the channel, it is the number of tranmistted signals vectors with the same channel matrix.
- double ber: Bit Error Rate.
- double ser: Symbol Error Rate.
- double bps: Bits per second processed (throughput).
- double itime: Time stamp when the simulation has been initiated.
- double etime: Time stamp when the simulation has been completed. When the simulation ends, it contains the time execution in seconds.
- mmp_dt_time det_time: It contains the execution time of the different stages of the detection.

A command is provided by the package to configure the simulation parameters:

mmp_simulation
new_simulation(int nT, int nR, int Nc, int Lch, double snr, double pot_s);

The mmp_dt_time structure is composed by five parameters, one for each step considered in the detection schemes depicted in Chapter 4, Fig. 4.1 and Chapter 5, Fig. 5.1.

```
typedef struct{
    double ord;
    double qr;
    double opt;
```

```
    double dt;
    double dac;
}mmp_dt_time;
```

where:

- double ord: Execution time in seconds of the channel ordering stage.
- double qr: Execution time in seconds of the QR decomposition stage.
- double opt: Execution time in seconds of the EEDC preprocessing stage.
- double dt: Execution time in seconds of the hard/soft detector stage.
- double dac: Execution time in seconds of the divide and conquer process stage when the Wintress framework is used.

6.5.6 Simulation Random Data

The user can use optionally the **mmp_data** structure to store the whole information related to the simulation without the need to manually allocate memory. This structure has de following form:

```
typedef struct{
    double *H_r;
    double *H_i;
    int *bits;
    double *s_r;
    double *s_i;
    double *y_r;
    double *y_i;
    double *sm_r;
    double *sm_i;
    double *ped;
    double *llr;
}mmp_data;
```

where:

- double *H_r: Real part of the channel matrices.
- double *H_i: Imaginary part of the channel matrices.
- int *bits: Bits to be transmitted.
- double *s_r: Real part of the signals to be transmitted.
- double *s_i: Imaginary part of the signals to be transmitted.
- double *sm_r: Real part of the estimated signals.
- double *sm_i: Imaginary part of the estimated signals.
- double *y_r: Real part of the received signals.
- double *y_i: Imaginary part of the received signals
- double *ped: Euclidean Distance of the estimated signal.
- double *llr: Log-likelihood ratio computed by the soft output detectors.

The following function creates the necessary variables to perform a simulation with MIMOPack. This function requires information of the different system, simulation and constellation sizes.

```
mmp_data simulation_data(int Nc, int Lch, int nR, int nT, int m);
```

6.6 MIMO Detection Functions

The first release of MIMOPack has been developed to achieve portability of codes across different computing environments (multi-core and GPUs). The library addresses the source code portability through common interfaces.

Hard and Soft output detectors have the same interface regardless of the execution platform. The function configuration is performed by passing the mmp_simulation, mmp_config, mmp_modulator and mmp_detector structures. The detector takes as input parameters the complex-valued channel matrix (H) and the received complex-valued signals vectors (y). The output of the hard detectors are the estimated complex-valued signal vectors (sm) and the associated euclidean distances (ED). On the other hand, the output of the soft detectors are the Log-likelihood ratios (LLR).

void **mmp_hard_detector**(mmp_simulation &sim, mmp_config conf, mmp_modulator mod, mmp_detector detector, double *ED, double *sm_r, double *sm_i, double *H_r, double *H_i, double *y_r, double *y_i);

void **hard_wintrees**(mmp_simulation &sim, mmp_config conf, mmp_modulator mod, mmp_framework framework, double *ED, double *sm_r, double *sm_i, double *H_r, double *H_i, double *y_r, double *y_i);

void **mmp_soft_detector**(mmp_simulation &sim, mmp_config conf, mmp_modulator mod, mmp_detector detector, double *LLR, double *H_r, double *H_i, double *y_r, double *y_i);

These functions call internally other specific functions that perform the preprocessing and detection stages depending on the user-selected input parameters. The list of all available functions with the relevant documentation can be found in section "Documentation" of the MIMOPack website.

6.7 Installation and Test

MIMOPack release is composed by two folders. The first one stores the library only for multicore processor (c-mimopack-1.0.tar.gz) and the other one takes care for GPUs or heterogeneous systems (multi-core/GPU) (cu-mimopack-1.0.tar.gz). The user must download the suitable version for his platform before starting the installation.

The first step for installing MIMOPack is to unpack the distribution tarball from "Download" website section. On a system with GNU tar installed, the package can be unpacked with the following command:

```
~$ tar zxf cu-mimopack-1.0.tar.gz
```

After unpacking the tarball you have to enter the newly created directory as

```
~$ cd cu-mimopack-1.0
```

You need to enter to the folder lib and run the installed script. You can specify in the first parameter the GPU architecture:

```
~$ cd lib
~/cu-mimopack-1.0/lib$ ./install sm_35
```

In order to determine whether the installation has been successfully carried out, the tarball contains a simple example of simulation. You have to enter into the test folder located within the main directory and run the make_test script.

```
~$ cd ../examples
~/cu-mimopack-1.0/examples$ ./make_test
```

If any compilation problem has been reported, you can run the test and use the library as

```
~/cu-mimopack-1.0/examples$ ./test
```

6.8 Example of simulation with MIMOPack

Let us consider a MIMO system with $n_T = 6$ transmitter and $n_R = 6$ receiver antennas and a certain signal-to-noise ratio (SNR). The input data stream is split equally into de n_T transmit antennas. The baseband equivalent model for this system is given by

$$\mathbf{y} = \mathbf{H}\mathbf{s} + \mathbf{v}, \tag{6.1}$$

where $\mathbf{s}$ represents the transmitted signal vector composed of the elements resulting of mapping sets of information bits to symbols belonging to a certain constellation $\mathcal{O}$ of size $M = 16$. Vector $\mathbf{y}$ denotes the received symbol

vector, and $\mathbf{v}$ is a complex additive white Gaussian noise vector. The detection problem in MIMO systems consists in determining the transmistted vector, denoted $\mathbf{sm}$, with the highest reliability. In practice, the detection problem is carried out by solving the following least squares problem

$$\mathbf{sm} = \arg \min_{\mathbf{s} \in \mathcal{O}^{n_T}} \|\mathbf{y} - \mathbf{Hs}\|^2. \tag{6.2}$$

To exemplify the use of the library, we show below an example for a simulation of the transmission of $N_c = 1000$ signals where the channel remain constant in the entire transmission. The detector is the ML Exhaustive and the parallelization is done by one GPU.

As we can see, there is an initial stage where some paremeters of the platform, modulator and simulation are configured (see lines 16-22). After that, we need to generate the random information bits to be transmitted. Next, these bits are multiplexed and mapped into constellations symbols. Furthermore it is necessary to generate the random channel values to simulate the conditions of the link. Note that, the input data *bits* and channel matrix H are in this example randomly generated. However, the user can provide their own input data. Then, we initialize the timer to assess the performance of the detector (MLE detector in line 29) with the configuration selected in the line 16. Finally, the user must save the simulation results to calculate some statistics (e.g. execution time, bit error rate, symbol error rate or throughput).

To evaluate the aforementioned proposed example, several simulations have been carried out to evaluate the complete MIMOPack functionality. The use-case scenarios represent different platform configurations provisioned in a High Performance computer today. We defined and used five execution types: (i) sequential and unoptimized; (ii) multi-core; (iii) sigle-GPU; (iv) multi-GPU; and (v) heterogenous execution modes. The experimental environment used to evaluate this example is the same as the one mentioned in Chapter 4, Section 4.1.3.

Figure 6.4 illustrates the speedup reached using the MLE MIMOPack detector with different platform configuration. The curve denoted by 80% (1 GPU) stands for the speedup obtained for the detection with $conf.pw = 80$. The number of signals computed by the GPU is selected as $N_c^{(gpus)} = 1000 \cdot \frac{80}{100} = 800$ and the number of signals computed by the multi-core

```
/*
 * mle_simulation.cu - example in C with mimopack.h library.
 * Performs the MLE detection of 1000 signals for a 6 x 6 MIMO system
 * with 16-QAM constellation using MIMOPack with one GPU.
 *
*/

#include <mimopack.h>

int main(int argc, char **argv){

1.  mmp_simulation sim;
2.  mmp_modulator  mod;
3.  mmp_config     conf;
4.  mmp_data       data;
5.  mmp_detector   detector;

6.  int n_cpus = 0;    // Number of OpenMP threads
7.  int n_gpus = 1;    // Number of GPUs
8.  int pw     = 100;  // GPU workload percentaje

9.  int nT = nR = 6;   // Number of transmitter/receiver antennas
10. int M = 16;        // Constellation size
11. int snr = 5;       // Signal to Noise Ratio in DB
12. int pot_s = 1;     // Power of the transmitted vector
13. int Nc  = 1000;    // Number of signals to be transmitted
14. int Lch = 1000;    // Assume constant channel
15. bool eedc = 1;     // Use EEDC

16. conf = set_mmp_config(num_gpus, omp_threads, work_gpu);
17. mod  = set_modulator(M, nT);
18. if(!check_config(conf))
19.     return 0;

20. int nbits = Nc*mod.m*nT;
21. sim      = new_simulation(nT, nR, Nc, Lch, snr, pot_s);
22. data     = simulation_data(Nc, Lch, nR, nT, mod.m);
23. detector = set_dt_mle("mle", "none", 1);

24. init_timer(sim);

25. generate_bits(data.bits, Nc*mod.m*nT);
26. random_awgn_channel(data.H_r, data.H_i, Lch, Nc, nT, nR);
27. qam_mapper_demux(mod, data.bits, Nc*mod.m*nT, data.s_r, data.s_i);
28. emulate_transmission(data.H_r, data.H_i, data.s_r, data.s_i,
                         data.y_r, data.y_i, sim.pot_n, Nc, Lch, nT, nR);
29. mmp_hard_detector(sim, conf, mod, detector, data.ped, data.sm_r, data.sm_i,
                      data.H_r, data.H_i, data.y_r, data.y_i);

30. save_simulation(sim, mod, data.s_r, data.s_i, data.sm_r, data.sm_i);

31. stop_timer(sim);

32. printf("Bit Error Rate          = %f\n", sim.ber);
33. printf("Time ordering           = %f\n", sim.dt_time.ord);
34. printf("Time QR decomposition   = %f\n", sim.dt_time.qr);
35. printf("Time EEDC preprocessing = %f\n", sim.dt_time.opt);
36. printf("Time detection          = %f\n", sim.dt_time.dt);
37. printf("Simulation time         = %f\n", sim.etime);

38. return 0;
}
```

Figure 6.3. Example of MIMOPack simulation: it performs the MLE detection of $N_c = 1000$ signals for a 6 x 6 MIMO system with 16-QAM constellation on one GPU.

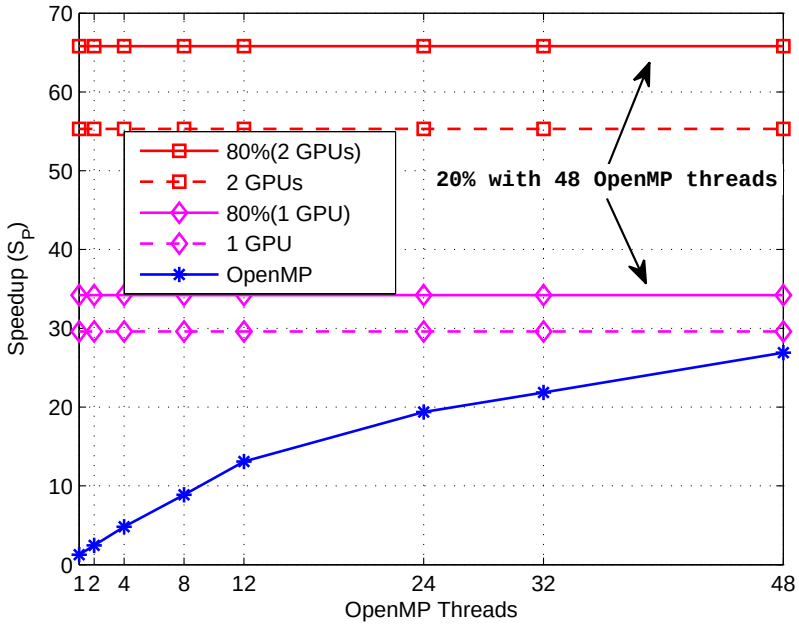


Figure 6.4. Speedup (S_p) comparison of the unoptimized MLE detector for the OpenMP and GPU implementations. A 6×6 MIMO system with 16-QAM constellation and $N_c = 1000$ for different library configurations is considered.

as $N_c^{(cpus)} = N_c - N_c^{(gpus)} = 200$ signals with 48 OpenMP threads. Similarly, curve denoted by 80% (2 GPU) detects simultaneosly 800 signals with 2 GPU. The experimental results show that the proposed heterogeneous implementation (80% (2 GPU)) outperforms the other methods. The distributed heterogeneous scheme allows to detect the signals up to 66 times faster than the unoptimized version. It can be seen from Fig. 6.4 that the OpenMP version is scalable when the number of threads increases and obtains up to 27 of speedup. In this case hyperthreading technology improves the performance when 48 threads are used. Note that, the use of two GPUs (see 2 GPUs curve in Fig. 6.4) does not imply a fifty per cent reduction in the execution time, this is because the transferences to the GPU memories do not overlap and the bandwidth of the PCI bus might be a bottleneck when it receives a high volume of data. Even so, the simultaneous use of two GPUs allows to decrease considerablelly the detection

time with respect the single GPU version (see 1 GPU curve in Fig. 6.4).

6.9 Conclusions

This chapter presents a first contribution to the MIMOPack library. This library is in constant evolution and needs a careful work of maintenance and development to bring new functionalities. Nevertheless, current library design and functionality allows easily to perform a wide range of simulations by setting properly the parameters depending on the MIMO scenario. This is especially useful for research centers and companies working in developing communication standards.

MIMOPack is a user-friendly portable software. Common interfaces enable to perform various tests on different platforms without requiring any changes of the code. Moreover, its modular and configurable structure allows a problem dependent configuration. It is designed to select the platform configuration among the available multi-core and GPU architectures, set the MIMO system parameters and simulate the transmission changing several meaningful parameters.

Experimental results show how MIMOPack can be used to run MIMO communication simulations up to 70 times faster using simultaneously a multi-core and multi-GPU platform. However, if the user does not have access to such a powerful machine he can also run his simulations in a faster way (up to 30x faster using a single GPU or 2 dodeca-core processors).

7 Conclusions

7 Conclusions

The overall aim of this research is to deepen into the MIMO Communications Systems algorithms and, specially, into the task of MIMO data detection. Researchers from different nature and with various objectives use these algorithms to study and develop technologies used in current MIMO communication systems. Usually these studies are expensive since require large hours of simulation and programming. The motivation of this research came from the necessity of developing a library containing, not only a wide range of receivers able to operate with different performance and complexity types, but also efficient versions of them in order to perform the detection in the shortest possible time; thus facilitating the work of these stakeholders.

This chapter summarizes the findings of this research work, revisiting the research objectives given in the introductory chapter. First, Section 7.1 reviews the contents of this study, outlining the main conclusions that were extracted from each chapter. Section 7.2 contains a list of works published during the course of candidature for the PhD degree. Additionally, recommendations for future research will be discussed in the final section.

7.1 Main Contributions

The main contribution of this research work has been the development of a high performance library, called MIMOPack, to implement the fundamental modules and processes carried during the transmission on a MIMO communication system. The goal of this work was to implement a set of parametrizable routines, highly optimized, with the aim to address certain needs that comercial libraries do not cover. This library is freely-available and can be downloaded from `http://www.inco2.upv.es/mimopack/index.html`. Current release contains a tree-search-based framework, 6 different hard output detectors, 4 soft output detectors, 2 matrix orderings and mathemathical and miscellaneous functions.

The final complexity of a MIMO system simulation depends heavily on three factors: the number of transmitter/receiver antennas, the constellation type and the detector used at the reception. For this reason, different types of detectors have been developed which can be classified into two general groups: hard-output and soft-output detectors, being the last ones those used in encoded transmissions. Each detector presents different performance in terms of Bit Error Rate (BER). Normally the most accurate (optimal) are those that need greater computational power. The design of the receivers not only has been devised to reduce the computational cost using the massively parallel computing capabilities of modern architectures (e.g GPUs), but different preprocessing modules have been integrated in order to accelerate them and increase the performance of those detectors that are considered not optimal.

The implementation of several MIMO hard and soft detection algorithms was described. Both, hard- and soft-output tree-search-based algorithms were implemented in OpenMP for multi-core architectures and CUDA for NVIDIA GPU devices even in a heterogeneous way. An efficient way to determine the Partial Euclidean Distances was additionally assessed and proposed to decrease the complexity of the detection process.

The computational times of the proposed efficient and parallel implementations were compared with their sequential execution and no optimized version in order to provide speedup results. Results were taken with a large number of transmitted signals and they showed that MIMOPack implemenations will improve and accelerate the simulations up to 70 times faster. Even including those using detectors that could not be properly addressed

without HPC machines (e.g ML Exhaustive detection).

A divide-and-conquer framework, called WinTress, for the design of thee-search based MIMO detectors was also proposed. This framework decomposes the initial detection tree into independent subtrees that can be simultaneously processed on a parallel architecture. This feature makes it a very suitable tool for several detectors especially for those with lower parallelism degree. Furthermore, the framework allows to fix the complexity of the detector by selecting a finite number of subtrees or reduce the number of them by using an initial radius. The performance results showed that the proposed framework achieves a good speedup even in a sequential implementation due to the Divide-and-Conquer scheme.

This work provides insight about the importance and the benefits of high performance computing in the scientific world, in particular for signal processing researchers, for whom MIMOPack will enable models and simulations with increased complexity. The ease of use, flexibility and portability are key features that determine the success of a software which can convert MIMOPack library in a interesting support tool for academic and industrial researchers.

7.2 List of Publications

A list of published work produced during the course of candidature for the degree is presented in what follows.

Refereed ISI Journals

- **C. Ramiro**, A.M. Vidal and A. Gonzalez. "MIMOPack: A High Performance Computing Library for MIMO communication systems". The Journal of Supercomputing, vol. 71, no. 2, pp. 751-760, February 2015.

- **C. Ramiro**, M.A. Simarro Haro, F. J.Martínez-Zaldivar, A.M.Vidal, A. Gonzalez, "A GPU implementation of an iterative receiver for energy saving MIMO ID-BICM systems". Journal of Supercomputing, vol. 70, no. 2, pp. 541-551, January 2014.

- **C. Ramiro**, S. Roger, A. Gonzalez, V. Almenar, A.M. Vidal, "Multi-Core Implementation of a Fixed-Complexity Tree-Search Detector for MIMO Communications". Journal of Supercomputing, vol. 65, no. 3, pp. 1010-1019, November 2013.

- S. Roger, **C. Ramiro**, A. Gonzalez, V. Almenar, A. M. Vidal, "Fully parallel GPU Implementation of a Fixed-Complexity Soft-Output MIMO Detector", IEEE Transactions on Vehicular Technology, vol. 61, no. 8, pp. 3796-3800, July 2012.

- S. Roger, **C. Ramiro**, A. Gonzalez, V. Almenar, A. M. Vidal, "An Efficient GPU Implementation of Fixed-Complexity Sphere Decoders for MIMO Wireless Systems", Integrated Computer-Aided Engineering, vol. 19, no. 4,pp. 341-350, September 2012.

Peer-reviewed non-ISI Journals

- M. Simarro, **C. Ramiro**, F.J. Martinez-Zaldivar, A.M. Vidal, A. Gonzalez, G. Piñero, V.M. Garcia, "Parallel Implementation Strategies for MIMO ID-BICM Systems", in Waves, vol. 5, pp.1889-8297, 2013.

- V.M. Garcia, A. Gonzalez, C. Gonzalez, F.J. Martinez-Zaldivar, **C. Ramiro**, S. Roger, A.M. Vidal, "The impact of GPU/Multicore in Signal Processing: a quantitative approach", in Waves, vol. 3, pp. 96-106, 2011.

Papers in International Conferences

- **C. Ramiro**, A.M. Vidal and A. Gonzalez, G. "Efficient Soft-Output Detectors: Multi-core and GPU implementations in MIMOPack Library". Proceedings of the 5th International Conference on Pervasive and Embedded Computing and Communication Systems, (PECCS), Angers, Loire Valley, France, February 2015.

- **C. Ramiro**, A.M. Vidal and A. Gonzalez, G. "MIMOPack: A HPC Library for MIMO Communication Systems". Doctoral Consortium

paper in the 5th International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS), Angers, Loire Valley, France, February 2015.

- **C. Ramiro**, A.M. Vidal and A. Gonzalez, G. "A HPC Library for MIMO communication systems: overview and prospectus". Proceedings of the 14th International Conference on Computational and Mathematical Methods in Science and Engineering, (CMMSE), Rota, Cadiz, Spain, June 2014.

- M. Simarro, **C. Ramiro**, F.J. Martinez-Zaldivar, A.M. Vidal, A. Gonzalez, G. "A parallel iterative MIMO receiver with variable complexity detectors". Proceedings of the 13th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE), Cabo de Gata, Almería, Spain, June 2013.

- **C. Ramiro**, J.J. López-Espín, D. Giménez, A.M. Vidal. "Two-Stage Least Squares Algorithms with QR Decomposition for Simultaneous Equations Models on Heterogeneous Multicore and Multi-GPU Systems", ICCS (International Conference on Computational Science), Procedia CS, vol. 9, pp. 2004-2007, 2012.

- **C. Ramiro**, S. Roger, A. Gonzalez, V. Almenar, A.M. Vidal, "Parallel Implementation of a Fixed-Complexity MIMO detector on a Multi-Core System". Proceedings of the 12th International Conference on Mathematical Methods in Science and Engineering (CMMSE), La Manga, Spain, July, 2012.

- F. Domene, S. Roger, **C. Ramiro**, G. Piñero, A. Gonzalez, "A Reconfigurable GPU Implementation for Tomlinson-Harashima Precoding", 37th International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Kyoto, Japan, March 2012.

- S. Roger, **C. Ramiro**, A. Gonzalez, V. Almenar, A.M. Vidal, "Rapid Prototyping of MIMO Detectors Using Graphic Processing Units", First Women's Workshop on Communications and Signal Processing, Banff, Canada, July 2012.

Papers in National Conferences

- **C. Ramiro**, A.M. Vidal, A. Gonzalez, L. Vergara, A. Salazar. "Towards a High Performance Computing Library for MIMO Communication Systems". XXIX Simposium Nacional de la Unión Científica Internacional de Radio (URSI), Valencia, Spain, September 2014.
- S. Roger, **C. Ramiro**, A. Gonzalez, V. Almenar, A.M. Vidal."On the use of graphic processing units for the efficient implementation of MIMO detector". XXVII Simposium Nacional de la Unión Científica Internacional de Radio (URSI), Elche, Spain, September 2012.
- F. Domene, S. Roger, **C. Ramiro**, G. Piñero, A. Gonzalez, "Efficient implementation of multiuser precoding algorithms on GPU for MIMO-OFDM systems", XXVII Simposium Nacional de la Unión Científica Internacional de Radio (URSI), Elche, Spain, September 2012.

7.3 Future Work

Following the investigations described in this thesis, the main lines of research that remain open are listed below:

- There are still different steps of MIMO communication systems simulation that can be accelerated by using multi-core and GPUs, such as channel matrix estimation, space-time codes and precoding techniques. In the short term it is intended to include new library modules that have already been developed but have not been included in this initial version:
  - Multi-User precoding techniques: Zero-Forcing, Tomlinson Harashima, Lattice Reduction Aided Tomlison Harashima and Enhanced Lattice Reduction Aided. The GPU implementatios have been reported in [86].
  - Low Density Parity Check Decoder which has been published in [81].

- Implementations of the detectors are made in a generic manner to allow the execution on any type of NVIDIA GPU architecture. However depending on the model installed on the user's platform some specific features (e.g. shared memory, nested parallelism or streams) may be considered to decrease the execution time. Therefore, the algorithms will be reimplemented wherever possible, in order to optimize them for each architecture.

- The library is optimized for its use on multicore and NVIDIA GPU systems. However we are studing the implementation of some detectors in new hardware architectures like Intel Many-Integrated-Core Xeon Phi or other types GPUs that are not manufactured by NVIDIA, leading to make implementations also in OpenCL.

- MIMOPack functions can be configured by selecting previously the parameters of the execution platform, however the user could not know in advance which are the optimal ones for obtain their results as fast as possible. Thus, an open research line is to develop an auto-optimization model in order to select the optimal parameters of specific implementations (e.g. block size of threads on the GPU) and also the optimum number and type of resources used to exploit the best execution performance of the platform.

- The library is entirely implemented from the ground up. Some of the auxiliary functions, such as vector and matrix operations, are already implemented in a very efficient way by other libraries (e.g BLAS, LAPACK, MAGMA). Furthermore, there exists a library IT++ which includes modules not implemented yet in MIMOPack. Hence, it would be interesting to combine MIMOPack with other existing libraries in order to obtain better functionality and speed.

- A software with these characteristics requires continuous modifications: to fix bugs, improve performance, add functionality, etc. Software maintenance is one of the most common and costly activities. Therefore, mechanisms for the evaluation, control and modifications will be developed.

7.4 Institutional Acknowledgements

This work has received financial support of the following projects:

- PROMETEO II: High Performance Computation and Communications and Applications in Engineering (PROMETEOII/2014/003. Generalitat Valenciana. Spain.)
- DISCOSOUND: Distributed and Collaborative Processing of Sound Signals: Algorithms, Tools and Applications (TEC2012-38142-CO4-01).
- PROMETEO: High Performance Computing Tools for solving Signal Processing Problems on Parallel Architectures (PROMETEO/2009/013. Generalitat Valenciana. Spain.)
- SMaGGic: Spatial audio systems based on Massive parallel processing of multichannel acoustic signals with General Purpose Graphics Processing Units (GPGPU) and Multi-cores (TEC2009-13741. Ministerio de Ciencia e Innovación. Spain).
- COPABIB: Automatic Development and Tuning of Parallel Libraries for Scientific Computation (TIN2008-06570-C04-02. Ministerio de Ciencia e Innovación. Spain).
- PAID-UPV: Development and Implementation of Computational Kernels for Software Defined Radio Platforms on Multicore/Manycore Architectures (PAID-05-10).

Bibliography

[1] G. J. Foschini and M. J. Gans, "On limits of wireless communications in a fading environment when using multiple antennas," *Wireless Personal Communications*, vol. 6, no. 3, pp. 311–335, 1998.

[2] A. J. Paulraj, D. A. Gore, R. U. Nabar, and H. Bolcskei, "An overview of MIMO communications-a key to gigabit wireless," *Proceedings of the IEEE*, vol. 92, no. 2, pp. 198–218, 2004.

[3] M. Jiang and L. Hanzo, "Multiuser MIMO-OFDM for next-generation wireless systems," *Proceedings of the IEEE*, vol. 95, no. 7, pp. 1430–1469, 2007.

[4] 3GPP TS 36.201 V10.0.0, "Evolved Universal Terrestrial Radio Access (e-utra); Physical layer general description," December 2010.

[5] F. Rusek, D. Persson, B. K. Lau, E. G. Larsson, T. L. Marzetta, O. Edfors, and F. Tufvesson, "Scaling up MIMO: Opportunities and challenges with very large arrays," *Signal Processing Magazine, IEEE*, vol. 30, no. 1, pp. 40–60, 2013.

[6] E. G. Larsson, O. Edfors, F. Tufvesson, and T. L. Marzetta, "Massive MIMO for next generation wireless systems," *arXiv preprint arXiv:1304.6690*, 2013.

[7] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A high-performance DSP

architecture for software-defined radio," *Micro, IEEE*, vol. 27, no. 1, pp. 114–123, 2007.

[8] C.-H. Yang *et al.*, "A multi-core sphere decoder VLSI architecture for MIMO communications," *IEEE Global Telecommunications Conference (GLOBECOM)*, pp. 1–6, 2008.

[9] D. Wu, J. Eilert, and D. Liu, "Implementation of a high-speed MIMO soft-output symbol detector for software defined radio," *Journal of Signal Processing Systems*, vol. 63, no. 1, pp. 27–37, 2011.

[10] NVIDIA, "GTC Presentation Archive, Signal and Audio Processing," online at: http://goo.gl/uqaQIj, 2015.

[11] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker, "Sora: high-performance software radio using general-purpose multi-core processors," *Communications of the ACM*, vol. 54, no. 1, pp. 99–107, 2011.

[12] T. Nylanden, J. Janhunen, O. Silvén, and M. Juntti, "A GPU implementation for two MIMO-OFDM detectors," in *International Conference on Embedded Computer Systems (SAMOS)*. IEEE, 2010, pp. 293–300.

[13] G. Falcão, V. Silva, and L. Sousa, "How GPUs can outperform ASICs for fast LDPC decoding," in *Proceedings of the 23rd International Conference on Supercomputing*. ACM, 2009, pp. 390–399.

[14] Y.-K. Chen, C. Chakrabarti, S. Bhattacharyya, and B. Bougard, "Signal processing on platforms with multiple cores: Part 1-Overview and methodologies [from the guest editors]," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 24–25, 2009.

[15] L. J. Karam, I. AlKamal, A. Gatherer, G. A. Frantz, D. V. Anderson, and B. L. Evans, "Trends in multicore DSP platforms," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 38–49, 2009.

[16] S. Tomov, R. Nath, D. Peng, and J. Dongarra, "MAGMA version 1.6 Users' Guide," online at: http://icl.cs.utk.edu/projectsfiles/magma/doxygen/, 2011.

[17] CULA, "CULA Programmer's Guide," online at: http://www.culatools.com/, 2014.

[18] T. MathWorks Inc., "Communication System Toolbox. Users Guide Version 6.5," online at: http://jp.mathworks.com/help/pdf_doc/comm/comm.pdf, 2014.

[19] IT++, "IT++ User's Guide," online at: http://itpp.sourceforge.net/4.3.1/users_guide.html, 2014.

[20] ETSI, "European Telecommunications Standards Institute," online at: http://www.etsi.org/index.php/membership.

[21] G. Caire, G. Taricco, and E. Biglieri, "Bit-interleaved coded modulation," *IEEE Transactions on Information Theory*, vol. 44, no. 3, pp. 927–946, 1998.

[22] G. J. Foschini, "Layered space-time architecture for wireless communication in a fading environment when using multi-element antennas," *Bell Labs Technical Journal*, vol. 1, no. 2, pp. 41–59, 1996.

[23] Z. Guo and P. Nilsson, "Algorithm and implementation of the K-best sphere decoding for MIMO detection," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 3, pp. 491–503, 2006.

[24] Q. H. Spencer, C. B. Peel, A. L. Swindlehurst, and M. Haardt, "An introduction to the multi-user MIMO downlink," *Communications Magazine, IEEE*, vol. 42, no. 10, pp. 60–67, 2004.

[25] J. J. Boutros, F. Boixadera, and C. Lamy, "Bit-interleaved coded modulations for multiple-input multiple-output channels," in *IEEE Sixth International Symposium on Spread Spectrum Techniques and Applications*, vol. 1. IEEE, 2000, pp. 123–126.

[26] J. R. Barry, E. A. Lee, and D. G. Messerschmitt, *Digital Communication*. Springer Science & Business Media, 2004.

[27] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger, "Closest point search in lattices," *IEEE Transactions on Information Theory*, vol. 48, no. 8, pp. 2201–2214, 2002.

[28] D. Micciancio and S. Goldwasser, *Complexity of lattice problems: a cryptographic perspective*. Springer Science & Business Media, 2002, vol. 671.

[29] J. Barry, E. Lee, and D. Messerschmitt, *Digital Communications*. United States: Springer Science & Business Media, 2003 (3rd Edition).

[30] R. Harris, D. M. Chabries, and F. Bishop, "A variable step (vs) adaptive filter algorithm," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 34, no. 2, pp. 309–316, 1986.

[31] T. Kailath, H. Vikalo, and B. Hassibi, *MIMO receive algorithms*. Cambridge University Press, 2005.

[32] B. Hassibi and H. Vikalo, "On the sphere-decoding algorithm i. expected complexity," *IEEE Transactions on Signal Processing*, vol. 53, no. 8, pp. 2806–2818, 2005.

[33] U. Fincke and M. Pohst, "Improved methods for calculating vectors of short length in a lattice, including a complexity analysis," *Mathematics of computation*, vol. 44, no. 170, pp. 463–471, 1985.

[34] C. P. Schnorr and M. Euchner, "Lattice basis reduction: improved practical algorithms and solving subset sum problems," *Mathematical programming*, vol. 66, no. 1-3, pp. 181–199, 1994.

[35] K. Su, C. Jones, and I. Wassell, "An automatic sphere decoder," *Submitted to IEEE Transactions on Information Theory*, 2004.

[36] W. Zhao and G. B. Giannakis, "Sphere decoding algorithms with improved radius search," *IEEE Transactions on Communications*, vol. 53, no. 7, pp. 1104–1109, 2005.

[37] R. A. Trujillo, "Algoritmos paralelos para la solución de problemas de optimización discretos aplicados a la decodificación de señales," Ph.D. dissertation, Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia, July 2009.

[38] V. M. Garcia-Molla, A. M. Vidal, A. Gonzalez, and S. Roger, "Improved Maximum Likelihood detection through sphere decoding combined with box optimization," *Signal Processing*, vol. 98, pp. 284–294, 2014.

[39] L. G. Barbero and J. S. Thompson, "Fixing the complexity of the sphere decoder for MIMO detection," *IEEE Transactions on Wireless Communications*, vol. 7, no. 6, pp. 2131–2142, 2008.

[40] M. Wu, Y. Sun, and J. R. Cavallaro, "Reconfigurable real-time MIMO detector on gpu," in *Conference Record of the Forty-Third Asilomar Conference on Signals, Systems and Computers*. IEEE, 2009, pp. 690–694.

[41] B. Wu and G. Masera, "Analysis on parallel implementations of fixed-complexity sphere decoder," *Science China Information Sciences*, vol. 56, no. 4, pp. 1–11, 2013.

[42] M. Wu, S. Gupta, Y. Sun, and J. R. Cavallaro, "A gpu implementation of a real-time MIMO detector," in *IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2009, pp. 303–308.

[43] M. S. Khairy, C. Mehlfuhrer, and M. Rupp, "Boosting sphere decoding speed through graphic processing units," in *European Wireless Conference*. IEEE, 2010, pp. 99–104.

[44] B. M. Hochwald and S. Ten Brink, "Achieving near-capacity on a multiple-antenna channel," *IEEE Transactions on Communications*, vol. 51, no. 3, pp. 389–399, 2003.

[45] M. R. Butler and I. B. Collings, "A zero-forcing approximate log-likelihood receiver for MIMO bit-interleaved coded modulation," *Communications Letters, IEEE*, vol. 8, no. 2, pp. 105–107, 2004.

[46] C. Studer, A. Burg, and H. Bolcskei, "Soft-output sphere decoding: Algorithms and VLSI implementation," *IEEE Journal on Selected Areas in Communications*, vol. 26, no. 2, pp. 290–300, 2008.

[47] MathWorks, "Communication System Toolbox. Examples," online at: http://es.mathworks.com/help/comm/examples.html, 2014.

[48] T. MathWorks Inc., "Parallel Computing Toolbox. Users Guide Version 6.5," online at: http://www.mathworks.com/help/pdf_doc/distcomp/distcomp.pdf, 2014.

[49] J. W. Eaton, D. Bateman, and S. Hauberg, *GNU octave*. Network Theory, 1997.

[50] E. Jones, T. Oliphant, and P. Peterson, "SciPy: Open source scientific tools for Python," *http://www. scipy. org/*, 2001.

[51] R. C. Whaley, "Atlas (automatically tuned linear algebra software)," in *Encyclopedia of Parallel Computing*. Springer Science & Business Media, 2011, pp. 95–101.

[52] MKL, "Intel Math Kernel Library. User's Guide," onlyne at: https://software.intel.com/sites/default/files/managed/9d/c8/mklman.pdf, October 2012.

[53] AMD, "AMD Core Math Library. User's Guide," onlyne at: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/acml.pdf, October 2012.

[54] Taeyoon Kim and Dr. Jeffrey G. Andrews, "MIMO-OFDM Design using LabVIEW," onlyne at: http://users.ece.utexas.edu/~jandrews/molabview.html, 2003.

[55] LabView Intiative at Wireless Networking and Communication Group, "MIMO Toolkit for LabView," onlyne at: http://users.ece.utexas.edu/~rheath/research/mimo/labview/MIMOToolkit.zip, 2003.

[56] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[57] A. Corp, "Heterogeneous System Architecture," online at: http://www.amd.com/es-xl/innovations/software-technologies/hsa, October 2014.

[58] The 500 most powerful commercially available computer systems, "Top500.org," online at: http://www.top500.org/lists/2014/11/, November 2014.

[59] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction," *Parallel Processing Letters*, vol. 18, no. 04, pp. 531–548, 2008.

[60] Y. Zhou, J. Liepe, X. Sheng, M. P. Stumpf, and C. Barnes, "GPU accelerated biochemical network simulation," *Bioinformatics*, vol. 27, no. 6, pp. 874–876, 2011.

[61] K. Zhang and J. U. Kang, "Real-time 4D signal processing and visualization using graphics processing unit on a regular nonlinear-k Fourier-domain OCT system," *Optics express*, vol. 18, no. 11, pp. 11 772–11 784, 2010.

[62] N. Kepler architecture, "NVIDIA Corporation," online at: http://www.nvidia.es/object/nvidia-kepler-es.html, 2012.

[63] A. Firepro Architecture, "Advanced Micro Devices (amd), inc." online at: http://www.amd.com/en-us/products/graphics/workstation/firepro-3d/9000, 2015.

[64] CUDA Toolkit Documentation, Version 6.5, "NVIDIA Corporation," online at: http://docs.nvidia.com/cuda/#axzz3TuVZFm5G, 2014.

[65] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems."

[66] N. GeForce GTX 750 Ti White Paper, "NVIDIA Corporation," online at: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf, 2014.

[67] NVIDIA Fermi architecture White Paper, "NVIDIA Corporation," online at: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidiafermicomputearchitecturewhitepaper.pdf, 2009.

[68] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[69] M. U. Guide, "The mathworks," *Inc., Natick, MA*, vol. 5, p. 333, 1998.

[70] The MathWorks Inc., "Create C Source MEX-File," online at: http://es.mathworks.com/help/matlab/matlab_external/standalone-example.html, October 2009.

[71] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, "A proposal for an extended set of Fortran basic linear algebra subprograms," *ACM Signum Newsletter*, vol. 20, no. 1, pp. 2–18, 1985.

[72] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen,

"LAPACK: A portable linear algebra library for high-performance computers," in *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 1990, pp. 2–11.

[73] AMD, "AMD Core Math Library. User's Guide," online at: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/acml.pdf, October 2014.

[74] Grey Ballard and James Demmel and Andrew Gearhart, "Communication bounds for heterogeneous architectures." LAPACK Working Note, Tech. Rep. 239, Feb. 2011.

[75] S. Roger, C. Ramiro, A. Gonzalez, V. Almenar, and A. M. Vidal, "Fully parallel GPU implementation of a fixed-complexity soft-output MIMO detector," *IEEE Transactions on Vehicular Technology*, vol. 61, no. 8, pp. 3796–3800, 2012.

[76] ——, "An efficient GPU implementation of fixed-complexity sphere decoders for MIMO wireless systems," *Integrated Computer-Aided Engineering*, vol. 19, no. 4, pp. 341–350, 2012.

[77] J. Jaldén, L. G. Barbero, B. Ottersten, and J. S. Thompson, "The error probability of the fixed-complexity sphere decoder," *Signal Processing, IEEE Transactions on*, vol. 57, no. 7, pp. 2711–2720, 2009.

[78] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to algorithms*. MIT press Cambridge, 2001, vol. 2.

[79] C. Ramiro, S. Roger, A. Gonzalez, V. Almenar, and A. M. Vidal, "Multicore implementation of a fixed-complexity tree-search detector for MIMO communications," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1010–1019, 2013.

[80] L. G. Barbero, T. Ratnarajah, and C. Cowan, "A low-complexity soft-MIMO detector based on the fixed-complexity sphere decoder," in *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*. IEEE, 2008, pp. 2669–2672.

[81] C. Ramiro, M. Á. Simarro, F. Martínez-Zaldívar, A. M. Vidal, and A. González, "A GPU implementation of an iterative receiver for energy saving MIMO ID-BICM systems," *The Journal of Supercomputing*, vol. 70, no. 2, pp. 541–551, 2014.

[82] C. Ramiro, S. Roger, A. Gonzalez, V. Almenar, and A. M. Vidal, "Parallel Implementation of a Fixed-Complexity MIMO detector on a Multi- Core System," in *Proceedings of the 12th International Conference on Mathematical Methods in Science and Engineering.* CMMSE, 2012.

[83] Zbigniew Siciarz, "Aquila: Open source DSP library for C++," online at: http://http://aquila-dsp.org/, 2010.

[84] T. P. Krauss, L. Shure, and J. N. Little, "Signal Processing Toolbox for use with MATLAB," 1994.

[85] C. Ramiro, A. M. Vidal, and A. Gonzalez, "MIMOPack: a high-performance computing library for MIMO communication systems," *The Journal of Supercomputing*, vol. 71, no. 2, pp. 751–760, 2015.

[86] F. Domene, S. Roger, C. Ramiro, G. Piñero, and A. Gonzalez, "A reconfigurable GPU implementation for Tomlinson-Harashima precoding," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on.* IEEE, 2012, pp. 1629–1632.