University of Alberta

Parallel Implementations of Detection Algorithms for MIMO Systems on The Graphics Processing Unit

by

Mengheng Jin

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer, Microelectronic Devices, Circuits and Systems

Department of Electrical and Computer Engineering

©Mengheng Jin

Spring 2014
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis and, except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

Dedicated to my beloved parents...

Abstract

In this thesis, the acceleration of detection algorithms for multiple-input multiple-output (MIMO) wireless systems is investigated. First we use the graphics processing unit (GPU), which provides thousands of parallel threads, to accelerate our detectors. The simulation environment for the parallel detectors is MATLAB with the Jacket library extension, which can modify conventional serial simulation codes to access the GPU and be executed in parallel. Comparisons between serial and parallel versions of different MIMO detectors are described in this thesis to determine how much speed-up that can be achieved from the parallelism. Furthermore, a parallel hybrid VBLAST-KBest detection algorithm is proposed that increases the accuracy beyond the conventional K-Best algorithm. The use of different forms of parallelism to speed-up matrix multiplication is investigated to provide insight into making the best use of the GPU. As a comparison, a multicore CPU acceleration using the parallel computing toolbox (PCT) is also briefly investigated.

~

Acknowledgement

First of all, I would like to express my sincere gratitude and respect to my supervisors Dr. Chintha Tellambura and Dr. Bruce Cockburn for their brilliant advices and limitless time they spent to help me during my M.Sc. program. With their professional knowledge and continuous and encouragement, I have learned a lot regarding how to analyse a problem, technical writing, presentation skills, etc. and I have learned even more from their great personalities. I feel fortunate to have this opportunity to study under their supervisions and I am sincerely grateful to them.

My thanks also goes to my M.Sc. examining committee Dr.Hai Jiang and Dr. Masum Hossain, for their time spent reading my thesis and providing valuable comments and advices. I am also grateful to the faculty and the staff of the Department of Electrical and Computer Engineering for their full support.

I would also like to thank Dr. Shuangshuang Han for her immense encouragement and valuable advices to my study, and Andrew Maier for his very helpful programming suggestions to my research. I also give many thanks to Prasanna and Russell for spending their time helping me to improve my oral presentation skills, and my labmates, including Jinghang and David for creating a pleasant lab environment.

My heartfelt and deepest gratitude goes to my beloved parents for their invaluable support and endless love throughout my life.

Thank you a lot!!!

~

Table of Contents

1 Introduction **1**

1.1 Background 1

1.2 Motivation 1

1.3 Outline of the Thesis 2

2 MIMO Systems **4**

2.1 Benefits of MIMO Technology 4

2.2 Technical Implementation of MIMO Systems 6

2.2.1 Spatial Multiplexing 6

2.2.2 Diversity Coding 6

2.2.3 Precoding 7

2.3 Characterization of MIMO Systems 7

2.3.1 Modulation Schemes 8

2.3.2 Signal-to-Noise Ratio 9

2.3.3 Bit Error Rate 9

2.3.4 Complexity 10

2.3.5 Diversity Order 10

2.3.6 Processing Speed 11

2.4 System Model 11

2.5 Overview of Linear MIMO Detection Methods 13

2.5.1 The Zero Forcing (ZF) Algorithm 13

2.5.2 The Minimum Mean Square Error (MMSE) Algorithm . . . 14

2.5.3 The Vertical BLAST (V-BLAST) Algorithm 15

2.5.4 Performance of the Linear Algorithms 16

2.6 Overview of the Sphere Detection Algorithm 17
2.6.1 The Fincke-Pohst (FP) Sphere Detection Algorithm 19
2.6.2 Schnorr-Euchner (SE) Enumeration 20
2.6.3 The K-Best Sphere Detection Algorithm 21
2.6.4 Pre-processing the Channel Matrix 22
2.6.5 Performance of the Sphere Detection Algorithms 24

3 Parallelism and the Graphics Processing Unit 28

3.1 Parallelism 28
3.1.1 Classification of Parallelism 28
3.1.2 The Limits of Parallelism - Amdahl's Law 31
3.2 The Graphics Processing Unit 32
3.2.1 Architecture of the GPU 33
3.2.2 The GPU Programming Model 35
3.3 Review of Past Parallel Implementations of MIMO Detectors 39

4 Parallel Implementation of MIMO Detection Algorithms on the GPU 44

4.1 Matrix Multiplication in Parallel 44
4.1.1 Experiment 1 for the for and gfor Looping Structures 45
4.1.2 Experiment 2 for the Serial and Parallel gfor Looping Structures 46
4.1.3 Experiment 3 for Merged Matrix Multiplication with Parallel gfor-loop 49
4.2 Models of Parallelism 52
4.3 Channel Generation on the GPU 54
4.4 Parallel Implementation of MIMO Detection Algorithms 54
4.4.1 Modification of Channel Inversion 55
4.4.2 Parallel Versions of the Linear MIMO Detection Algorithms 57
4.4.3 The Parallel V-BLAST Algorithm 61
4.4.4 Parallel V-BLAST with Real and Imaginary Components . . 63
4.4.5 The Parallel K-Best Algorithm 66
4.4.6 The Fully Enumerated K-Best Algorithm 66

4.4.7 The Parallel V-BLAST with K-Best Algorithm 69

5 Parallel Implementation of MIMO Detection Algorithms Using the Parallel Computing Toolbox in MATLAB 73

5.1 Parallelism in MATLAB 73

5.2 Matrix Multiplication Using the Parallel Computing Toolbox 75

5.3 Parallelism Models and the Performance Achieved Using the Parallel Computing Toolbox 77

6 Conclusions 81

6.1 Contributions 81

6.2 Future Work 82

Bibliography 84

A Source Codes for Serial MIMO Detection Algorithms 89

A.1 Main Function for Different Detection Algorithms 89

A.2 Maximum Likelihood (ML) Detection Algorithm 90

A.3 Zero Forcing (ZF) Detection Algorithm 92

A.4 Minimum Mean Square Error (MMSE) Detection Algorithm 92

A.5 V-BLAST Detection Algorithm 93

A.6 Fincke-Pohst (FP) Sphere Detection Algorithm 94

A.7 Schnorr-Euchner (SE) Sphere Detection Algorithm 96

A.8 K-Best Sphere Detection Algorithm 99

B Source Codes for Parallel MIMO Detection Algorithms 103

B.1 Main Function for Different Detection Algorithms 103

B.2 New Matrix Inverse Function 105

B.3 Zero Forcing (ZF) Detection Algorithm Parallel Version 107

B.4 Minimum Mean Square Error (MMSE) Detection Algorithm Parallel Version 108

B.5 V-BLAST Detection Algorithm Parallel Version 109

B.6 Parallel V-BLAST Detection Algorithm 111

B.7 K-Best Sphere Detection Algorithm Parallel Version 116
B.8 Parallel V-BLAST Detection Algorithm with Real and Imaginary Components 119
B.9 Fully Enumerated K-Best Detection Algorithm 125
B.10 Parallel VBLAST-K-Best Detection Algorithm 132

List of Tables

4.1 Matrix multiplication mimes (in seconds) for different looping (for and gfor) structures 46
4.2 Matrix multiplication times (in seconds) for serial and different degrees of parallel versions 48
4.3 Matrix multiplication times (in seconds) for the merged matrix with parallel gfor-loop structure 51
4.4 Comparison of matrix inverse runing times (in seconds) using built-in function "inv" and new function "NewInverse" 56
4.5 Running times (in seconds)comparison of MIMO detection algorithms with the serial and different parallel versions 58
4.6 The most time consuming operations for the MIMO detection algorithms 60

5.1 Matrix multiplication times (in seconds) using the for, gfor and parfor loops 76
5.2 Running times (in seconds) comparison of MIMO detection algorithms with the serial and different parallel versions 79

List of Figures

2.1 Multiple-antenna structures 5

2.2 Constellation diagram for rectangular 16-QAM. 9

2.3 Performance of three linear MIMO detection algorithms (ZF, MMSE and V-BLAST) compared to the optimal detection method (ML detection) for $M_t = M_r = 4$, and 16-QAM modulation. Each data point represents at least 100 detection errors. 16

2.4 Search tree model for successive symbol detection 18

2.5 Performance of three detection algorithms (FP, SE, K-Best when $K = 6$) with and without preprocessing and for a $M_t = M_r = 4$, 16-QAM MIMO system. Each data point represents at least 100 detection errors. 24

2.6 Effects of preprocessing for a $M_t = M_r = 4$, 16-QAM MIMO system when $K = 1, 4$, and 16. Each data point represents at least 100 detection errors. 25

2.7 Complexity of three detection algorithms (FP, SE, K-Best when $K = 6$) for a $M_t = M_r = 4$, 16-QAM MIMO system. 26

3.1 Amdahl's Law. The speed-up of a program executed in parallel by different numbers n of multiprocessors with different degrees of parallelism. 32

3.2 Memory structure of a GPU [1] 34

3.3 If-then-else construct replaced with a multiplied condition factor. . . 38

4.1 Distribution of 4-PAM symbols, additive noise and MIMO channel coefficients 55

4.2 Performance of conventional V-BLAST, Parallel V-BLAST and Real-Imaginary component V-BLAST for a $M_t = M_r = 4$, 16-QAM MIMO system. 65

4.3 Performance of the K-Best and the fully enumerated K-Best for a $M_t = M_r = 4$, 16-QAM MIMO system. 67

4.4 Algorithmic structure of the parallel V-BLAST with K-Best algorithm. 70

4.5 Performance of the VBLAST-KBest hybrid MIMO detector for a $M_t = M_r = 4$, 16-QAM MIMO system. 72

5.1 MATLAB parallel computing toolbox worker pool structure 74

List of Abbreviations

Abbreviation	Definition
3GPP	3rd-generation partnership project
4G	4th generation
ASK	amplitude shift keying
BER	bit error rate
BLAST	Bell Laboratories layered space-time
BP	Babai point
CPU	central processing unit
CUDA	compute unified device architecture
FP	Fincke-Pohst
FSK	frequency shift keying
GPU	graphics processing unit
IEEE	Institute of Electrical and Electronics Engineers
ISI	inter-symbol interference
LTE	long term evolution
MIMD	multiple-instruction multiple-data
MIMO	multiple-input multiple-output
MISD	multiple-instruction single-data
MISO	multiple-input single-output
ML	maximum likelihood
MMSE	minimum mean square error
OFDM	orthogonal frequency division multiplexing
OpenCL	Open computing language
PCT	parallel computing toolbox

Abbreviation	**Definition**
PED	partial Euclidean distance
PSK	phase shift keying
QAM	quadrature amplitude modulation
SD	sphere detector
SE	Schnorr-Euchner
SER	symbol error rate
SIMD	single-instruction multiple-data
SIMO	single-input multiple-output
SISD	single-instruction single-data
SISO	single-input single-output
SM	stream multiprocessor
SNR	signal-to-noise ratio
WiMAX	Worldwide interoperability for Microwave Access
ZF	zero forcing

List of Symbols

Notation	Definition
$\lvert a \rvert$	absolute value of scalar a
$\lceil a \rceil$	the smallest integer greater than or equal to a
$\lfloor a \rfloor$	the largest integer less than or equal to a
$\Re(a)$	the real component of (complex) scalar a
$\Im(a)$	the imaginary component of (complex) scalar a
$\mathbf{A}$	real-valued matrix **A**
$\tilde{\mathbf{A}}$	complex-valued matrix **A**
$\hat{\mathbf{A}}$	detected matrix **A**
$\mathbf{A}(i,j)$	i-th element of the j-th column of matrix **A**
$\lVert \mathbf{A}-\mathbf{B} \rVert_F^2$,	Euclidean distance between **A** and **B**
$\mathbf{A}^{-1}$	inverse of square matrix **A** (for $m = n$)
$\mathbf{A}^{\dagger}$	Moore-Penrose pseudo inverse of matrix **A** (for $m = n$)
$\mathbf{A}^H$	conjugate transpose of matrix **A**
$\mathbf{A}^T$	transpose of matrix **A**
$\mathbf{I}_n$	identity matrix of rank n
$\min(a,b)$	minimum of scalars a and b
$\arg\min_i(\mathbf{A})$	index i corresponding to the smallest element a_i of set **A**
$\log(\cdot)$	natural logarithm
$\log_2(\cdot)$	logarithm to base 2
$\sum_{i=i}^{m} x_i$	summation over all x_i for $i = 1, 2, \ldots, m$
∞	infinity
$\lim_{x\to a} f(x)$	the limit of function $f(x)$ as x tends to a
$\int_a^b f(x)dx$	the integral of function $f(x)$ from a to b
$\Gamma(\cdot)$	Gamma function

Chapter 1

Introduction

1.1 Background

Wireless communications provides key infrastructure used in modern daily life. The convenience of wireless allows us to use cellular telephones and wirelessly connected computers almost everywhere in towns and cities and along major transportation corridors. However, since the limited wireless bandwidth can not cope with the rapidly increasing user traffic, multiplexing technology has become an essential way for better exploiting limited channel resources. A popular method to increase the wireless capacity within a fixed bandwidth is multipath propagation among one or more transmitting and receiving antennas. In this thesis, we consider the multiple-input multiple-output (MIMO) system which makes full use of multiple antennas at both the transmitter and receiver ends of the channel to achieve significant improvements in wireless system performance.

1.2 Motivation

Wireless signals propagate from the transmitter to the receiver through the radio channel. However, because the radio channel has various inevitable sources of noise and fading attenuation, the received signal is distorted and detection errors can occur at the receiver. In a MIMO channel, each receiver antenna receives superimposed copies of all of the transmitted signals In order to recover (detect) transmit data from the received signal with a lower bit error rate (BER), researchers have already investigated many ways to improve the performance of the MIMO detector.

MIMO detector employs maximum likelihood principles to recover the transmit data. Many MIMO detection algorithms have been proposed that can approach the statistically optimal performance of maximum likelihood detection [2]. However, the high computational complexity of these algorithms has made them unsuitable for widespread adoption in practical MIMO receiver designs.

Hardware parallelism is now provided in various ways in the instruction sets and architectures of most computers [3]. Parallel computing exploits the fact that large problems can often be divided into smaller computations, which can then be solved concurrently to reduce the total required running time. Traditionally, to solve a problem, an algorithm is designed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit (CPU) on one computer. Only one instruction may execute at a time. Parallel computing, on the other hand, uses multiple processing elements to solve a problem simultaneously. This is accomplished by splitting the problem into several independent parts so that each processing element can execute simultaneously in parallel with the others. The processing elements could be diverse and could include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above. In this research, we investigate different ways to exploit the forms of hardware parallelism available in the simulation of a MIMO system with the objective of measuring and maximizing performance and efficiency. Insights obtained while implementing a parallel MIMO simulation could lead to improve parallel MIMO detectors of benefit to wireless communications equipment.

The main objective of this project is to find a better way to implement the parallelism, either on the multicore CPU or the GPU subsystem, and to achieve significant acceleration in some of the MIMO detection algorithms.

1.3 Outline of the Thesis

The rest of the thesis is organized as follows:

Chapter 2 provides an introduction to MIMO wireless technology. This chapter

describes the major MIMO detection algorithms including several variants of the sphere detector (SD) algorithm. A brief comparison of these algorithms is provided at the end of the chapter.

Chapter 3 introduces the graphics processing unit (GPU) and the application of parallel GPU-based computing for MIMO detection. There are several ways to exploit GPU parallelism. The first parallel programming environment evaluated in this thesis is the Jacket library extension of the MATLAB environment.

Chapter 4 describes the details of the parallel MIMO detector implementations on the GPU. Since all of the simulated data are created initially on the CPU in serial fashion, it is important to have an efficient method to map the calculations efficiently onto the parallel GPU hardware. There are often limitations imposed on the algorithms by the hardware parallelism. For example, one might be required to synchronize the same kinds of arithmetic operations on parallel streams of data. Several challenges are addressed in this chapter. At the end of the chapter, the simulation and experimental evaluation of the developed parallel MIMO detection algorithms are discussed.

Chapter 5 compares the performance of parallel computation on the GPU and the multiple cores of the CPU. The parallel computing toolbox (PCT) in MATLAB is used to implement the parallelism on the multicore CPU. Several detection algorithms introduced in Chapters 2 and 4 are run and compared to find a better way to exploit the different kinds of hardware parallelism.

Chapter 6 includes the conclusions arising from the research presented in this thesis and gives recommendations for future work.

∼

Chapter 2

MIMO Systems

2.1 Benefits of MIMO Technology

MIMO technology is now widely used in wireless communication standards. Depending on the number of the antennas at both the transmitter and receiver ends, there are three special cases of MIMO include single-input single-output (SISO), single-input multiple-output (SIMO), and multiple-input single-output (MISO). These four systems are illustrated in Fig. 2.1.

For $M_t \geqslant 1$ transmit antennas and $M_r \geqslant 1$ receiver antennas, the data streams can be propagated in parallel through the capacity equivalent of $\min(M_t, M_r)$ different channels. For example, for a rich scattering MIMO channel (i.e., a channel where the rows and columns of the channel matrix are linearly independent), the channel capacity [4] has been shown to be:

$$\mathrm{C} = \min(M_t, M_r)\mathbf{B}\log_2(1+\rho) \qquad bps \tag{2.1}$$

where ρ is the average signal-to-noise ratio (SNR) at the receiver. Eq. (2.1) shows that when the signal bandwidth **B** and SNR ρ are fixed, the channel capacity can be linearly increased by increasing the number of antennas as long as the channel remains rich scattering. Sufficiently rich scattering is required to allow signal processing to disentangle the multiple transmitted signals in the MIMO receiver. Equivalently, Eq. (2.1) also indicates that the spectral efficiency (bits per second per hertz of bandwidth), which indicates the number of users that can be simultaneously supported on a limited frequency bandwidth, can be increased by spreading the total transmitted power over the available antennas to achieve an improvement

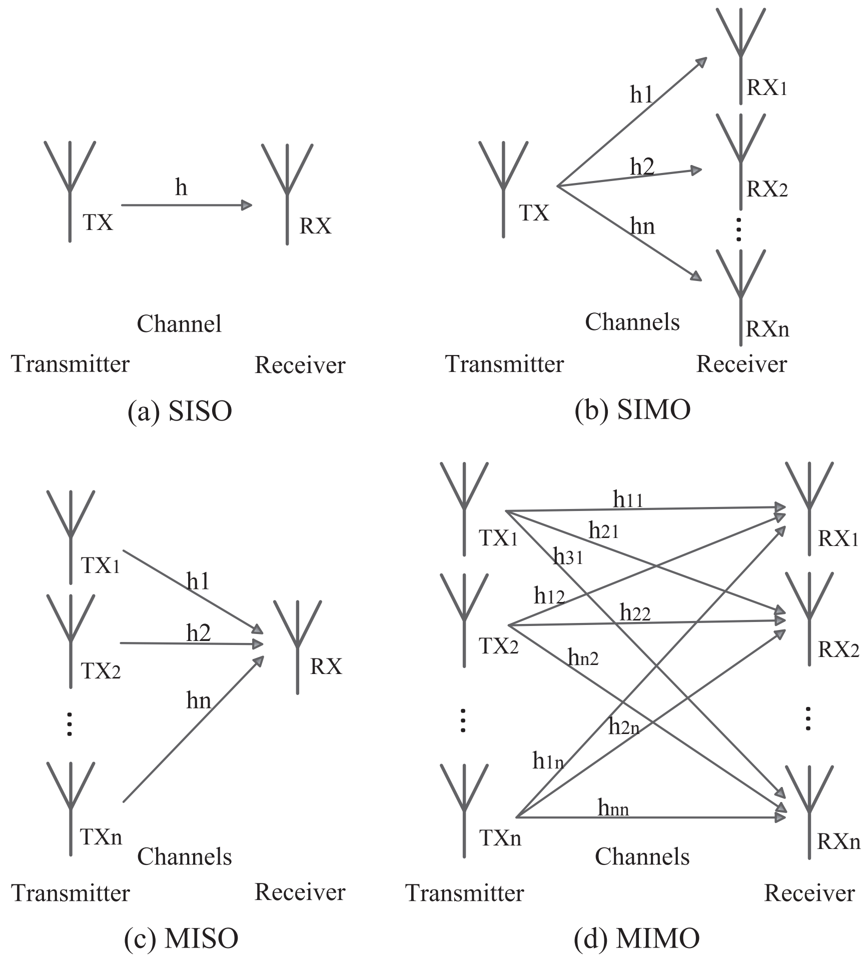

Figure 2.1: Multiple-antenna structures

without consuming additional bandwidth. Furthermore, by employing more antennas at the receiver side, one can reduce the vulnerability to channel fading to improve the link reliability. Fading is the sometimes severe attenuation of the signal strength at a receiver antenna caused by destructive interference among the multiple superimposed received signals. In general, MIMO technology also can ensures the independence of each signal copy from different transmitters to achieve a lower error rate at the receiver. Because of these properties, MIMO technology plays an important role in many modern wireless communication standards, for example, in IEEE 802.11n (Wi-Fi) [5], 4G [6], the 3rd-generation partnership project (3GPP) long term evolution (LTE) [7] and IEEE 802.16 (WiMAX) [8].

2.2 Technical Implementation of MIMO Systems

2.2.1 Spatial Multiplexing

Spatial multiplexing is a common MIMO scenario (Fig. 2.1 (d)). Its main principle is to first split the data stream into several independent sub streams and then to transmit them from different transmitter antennas within the same frequency range. Compared to a conventional SISO system, spatial multiplexing improves the throughput rate to achieve much higher frequency spectrum utilization. If the MIMO channel between the transmitter and receiver arrays provides sufficient diversity due to the rich scattering in the channel, the receiver can detect these parallel data streams reliably. Spatial multiplexing technology can be applied successfully at the receiver without knowing and exploiting the channel state information at the transmitting side. The Bell Laboratories Layered Space-Time (BLAST) system [9], developed by Foschini and other researchers at Bell Labs, was an early example of practical spatial multiplexing technology.

2.2.2 Diversity Coding

Diversity coding may be seen as transmitting multiple independent copies of the same signal to the multiple receivers over relatively independent different physical paths through space. These copies are then combined into one output signal at the

receiver. This combination step effectively reduces the effects of channel fading affecting any one of path to ensure a robust system by increasing the effective number of independent channels. To maximize the signal diversity, space time coding [10] is used in MIMO systems to ensure that all transmitted data are sent out on all transmitter antennas and then received on all receiver antennas. A suitable space-time decoder is required at the receiver to efficiently recover the data from the signals obtained from all receiver antennas.

2.2.3 Precoding

In general, precoding applies several transformations on the signals at the transmitter to simplify the detection at the receiver. The premise of precoding is that when channel state information is known at the transmitter, then during the precoding, appropriate phase and gain weighting can be applied to the transmitted signals to reduce multipath fading effects suffered by the signals during propagation. Precoding can also be seen as multi-stream beamforming, which also attempts to reduce the interference from the transmission environment.

As we can see, spatial multiplexing sends the different data streams over effectively parallel channels over same propagation path while spatial diversity transmits with greater reliability the same information via different channels. Thus there exists a trade-off between the system capacity and reliability. The combination of MIMO technology and orthogonal frequency division multiplexing (OFDM) [11] in many broadband wireless standards is a good example of how to make full use of these two strategies.

2.3 Characterization of MIMO Systems

To measure the performance of a MIMO system, we consider the following variables.

2.3.1 Modulation Schemes

Standard modulation techniques include phase shift keying (PSK) modulation, frequency shift keying (FSK) modulation, amplitude shift keying (ASK) modulation, quadrature amplitude modulation (QAM). In this thesis, we focus on QAM modulation, which is widely used in the highest-capacity broadband wireless systems.

In QAM, the digital bit stream modulates the amplitudes of two orthogonal carriers (on sine and cosine) of the same frequency. Because QAM makes full use of both the amplitude and phase of two orthogonal carriers, the bandwidth efficiency is increased. A QAM constellation diagram is a two-dimensional scatter plot of a digital modulated signal in the complex plane. In QAM, if a suitable constellation size is chosen, it is possible to achieve relatively high spectral efficiencies, limited only by the signal-to-noise ratio and the effects of distortion and fading in the communications channel. The constellation points are usually packed within a square or rectangular grid with equal vertical and horizontal spacing. Because data in digital communications is in binary format, it is convenient that the number of points in the grid be a power of 2 (such as 2, 4, 8, . . .). Each point maps a group of data bits (e.g., 2, 4, 8, . . .) forming a code word to a unique transmitted complex symbol in the transmitter. The constellation diagram of 16-QAM, which is used in this research, is shown in Fig. 2.2.

Following standard practice, the Gray code [12] scheme was used to map code words to constellation points. Adjacent constellation points correspond to code words that differ in exactly on bit. In 16-QAM, the data is transmitted using 4-bit symbols. So during the transmission, the number of data bit errors is minimized when symbol detection errors occur.

According to Fig. 2.2, the data stream is mapped to a complex plane by demultiplexing them into real and imaginary substreams, converting consecutive bit pair "00" to "-3", "01" to "-1", "10" to "+1", "11" to "+3". Note that each complex-valued symbol encodes 4 bits (Fig. 2.2 where the axis labels I and Q stand for the real and imaginary part, respectively). It is possible to transmit more bits per symbol by applying a higher-order constellation. However, higher-order QAM constellations mean that the constellation points are more closely spaced together and

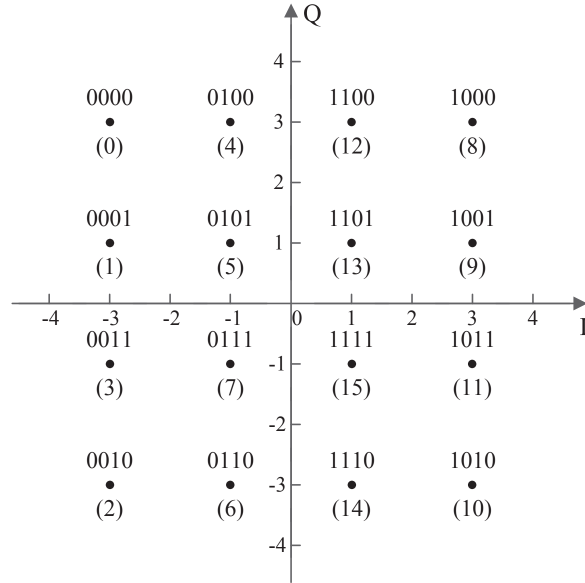

Figure 2.2: Constellation diagram for rectangular 16-QAM.

are thus more susceptible to noise and other signal corruptions, possibly leading to incorrectly detected symbols and hence to bit errors.

2.3.2 Signal-to-Noise Ratio

The SNR is a widely used measure of signal quality in communications engineering. In general, the SNR is the ratio of the signal power to the noise power. The SNR is also usually expressed in logarithmic form in decibels (dB) because of the wide dynamic range of typical SNRs. A higher SNR implies that the system should be less influenced by background noise and signal distortion. In a MIMO system, it is easier to detect the signal at the receiver side in a higher SNR environment, and therefore improving the accuracy of the detector output. In that scenario, our design goals are to achieve the highest possible detection accuracy with the least signal processing computation at the receiver.

2.3.3 Bit Error Rate

As we described above, the accuracy of the detector is extremely important to the overall performance of MIMO system. Detection accuracy is measured by the bit error rate (BER). The BER is computed by taking the number of mis-detected bits

compared to the number of originally transmitted bits:

$$\text{BER} = \frac{N_{error}}{N_{total}}. \tag{2.2}$$

In Eq. (2.2), N_{error} denotes the total number of error bits seen at the receiver, and N_{total} denotes the total number of the bits that were transmitted. When using QAM it is simpler and thus common practice to calculate the related quantity called the symbol error rate (SER) instead of the BER. In 16-QAM, there are 16 possible symbols and the SER can be obtained by replacing the N_{error} and N_{total} with the number of error symbols and the total number of transmitted symbols respectively. For general M-QAM, the SER will be roughly $\log_2 M$ times the corresponding BER because one symbol detection error can cause more than one errored bit.

2.3.4 Complexity

The computational complexity of the MIMO detector is important. Time complexity results are expressed as the number of executed representative CPU operations (e.g., adds and multiplies) and memory operations (e.g., reads and writes) in the algorithm. Space complexity results express the memory storage requirements (e.g., maximum required number of stored bytes). Especially in the sphere detector algorithms, the nodes of a search tree are systematically visited as a solution (e.g., the detected symbol vector) is progressively constructed. Since several possible tree searching algorithms can be applied to find the optimal path, there is a trade off between the efficiency (i.e., average number of nodes visited) and symbol vector detection accuracy. To compare the complexity among different algorithms, we count up the number of nodes which are visited during the tree search. Thus, the mean number of visited nodes can also be used as a measure of time complexity.

2.3.5 Diversity Order

The definition of diversity order is related to the effective number of statistically independent fading channels between the transmitter and the receiver. If the fading in each transmit-receive pair of antennas is statistically independent, the diversity

order of the MIMO channel can be shown to be:

$$d = M_t \times M_r. \tag{2.3}$$

When the SNR and the error probability are measured experimentally in a system simulation, the diversity order can be shown to be [13]:

$$d = -\lim_{\rho\to\infty} \frac{\log \mathrm{P_e}(\rho)}{\log \rho}, \tag{2.4}$$

where ρ denotes the signal-to-noise ratio and $\mathrm{P_e}(\rho)$ denotes the error probability, which is taken to be the SER in this thesis. In this expression, the diversity order can be seen to be the magnitude of the slope of the error probability vs. SER curve on a log-log plot. This implies that for the same SNR, the use of a higher diversity order detector can achieve a lower SER. MIMO diversity coding mentioned in Section 2.2.2 has been designed to maximize the diversity order. In contrast, spatial multiplication does not attempt to maximize the diversity order but instead maximizes the data rate.

2.3.6 Processing Speed

Processing speed is affected by many factors, including most obviously the running time, which gives the required number of CPU or GPU instructions. The hardware of the processing device also plays an important role in determining the time complexity cost per bit. In this research, we investigate alternative ways of improving MIMO detection algorithms by converting the data and node searching algorithms into parallel form to better exploit the characteristics of the available parallel hardware. Whenever a group of data values can be processed at the same time, the computation time should be reduced compared to the serial computation.

2.4 System Model

As we described above, if the signal information is propagated between $M_t \geqslant 1$ transmit antennas and $M_r \geqslant 1$ receive antennas over a frequency non-selective fading channel, then the MIMO system model can be expressed as:

$$\tilde{\mathbf{y}} = \tilde{\mathbf{H}}\tilde{\mathbf{s}} + \tilde{\mathbf{n}}, \tag{2.5}$$

where $\tilde{\mathbf{y}} = [\tilde{y}_1 \ \tilde{y}_2 \ \cdots \ \tilde{y}_{M_r}]^T$ is the M_r-element received signal vector, where the operation $[\cdot]^T$ is the transpose of a vector/matrix, $\tilde{\mathbf{H}}$ denotes the $M_r \times M_t$ channel matrix, where following standard practice the elements $\tilde{h}_{ij}$ of $\tilde{\mathbf{H}}$ are independent and identical complex, zero-mean and Gaussian-distributed, $\tilde{\mathbf{s}} = [\tilde{s}_1 \ \tilde{s}_2 \ \cdots \ \tilde{s}_{M_t}]^T$ denotes the M_t-element transmitted signal vector, whose elements $\tilde{s}_i$ represent independent symbols drawn from a complex constellation such as 4-QAM, 16-QAM, 64-QAM, and $\tilde{\mathbf{n}}$ is an $M_r \times 1$ vector of independent white Gaussian noise samples. In this thesis, we make the common assumption that $M_t = M_r$ and that the channel $\tilde{\mathbf{H}}$ is perfectly estimated at the receiver as a result of a suitable training mechanism.

In Eq. (2.5), all the variables are complex. However, an equivalent real-valued system can be expressed as [14]:

$$\mathbf{y} = \mathbf{H}\mathbf{s} + \mathbf{n}, \tag{2.6}$$

i.e.,

$$\begin{bmatrix} \Re(\tilde{\mathbf{y}}) \\ \Im(\tilde{\mathbf{y}}) \end{bmatrix} = \begin{bmatrix} \Re(\tilde{\mathbf{H}}) & -\Im(\tilde{\mathbf{H}}) \\ \Im(\tilde{\mathbf{H}}) & \Re(\tilde{\mathbf{H}}) \end{bmatrix} \begin{bmatrix} \Re(\tilde{\mathbf{s}}) \\ \Im(\tilde{\mathbf{s}}) \end{bmatrix} + \begin{bmatrix} \Re(\tilde{\mathbf{n}}) \\ \Im(\tilde{\mathbf{n}}) \end{bmatrix} \tag{2.7}$$

where $\Re(\cdot)$ and $\Im(\cdot)$ represent the real and imaginary parts of the corresponding elements of complex vectors and matrices. As a result, when going to a real-valued system, the dimensions of $\mathbf{y}$, $\mathbf{H}$, $\mathbf{s}$, and $\mathbf{n}$ grow to $2M_r \times 1$, $2M_r \times 2M_t$, $2M_t \times 1$, and $2M_r \times 1$, respectively.

The objective of MIMO detection is to find the signal vector that minimizes the Euclidean distance between the predicted noise-free signal vector $\mathbf{Hs}$ and the received vector $\mathbf{y}$ in the presence of the Gaussian noise $\mathbf{n}$ [15]. Statistically optimal performance is obtained using the maximum likelihood (ML) detection rule, i.e.,

$$\hat{\mathbf{s}} = \arg\min_{\mathbf{s}\in\Omega} \|\mathbf{y} - \mathbf{H}\mathbf{s}\|^2, \tag{2.8}$$

where $\hat{\mathbf{s}}$ is the detected signal vector and Ω stands for the set of the real entries along one dimension in the constellation, e.g., $\Omega = \{-3, -1, 1, 3\}$ if we are considering 16-QAM. $\|\cdot\|^2$ denotes the sum of the squares of the corresponding elements. For convenience, we also define $M_c = \sqrt{M}$ to be the equivalent real-valued constellation size of M-QAM (i.e., $M_c = 4$ in 16-QAM).

2.5 Overview of Linear MIMO Detection Methods

In a MIMO system, linear detection algorithms are widely used methods at the receiver. In a linear detection algorithm, the computational complexity grows linearly in the number of antennas. First the received signal vector undergoes a linear transformation by being pre-multiplied by a conditioning matrix (e.g., matrix computed using the Zero-Forcing or MMSE criteria), then the resulting signals quantized it to the closest constellation points. In these algorithms, we obtain the complex form of the signal vector $\tilde{\mathbf{y}}$ using the system model from Eq. (2.5). That is:

$$\tilde{\mathbf{y}} = \left[\tilde{\mathbf{h}}_1 \; \tilde{\mathbf{h}}_2 \; \cdots \; \tilde{\mathbf{h}}_{M_t}\right] \begin{bmatrix} \tilde{s}_1 \\ \vdots \\ \tilde{s}_{M_t} \end{bmatrix} + \tilde{\mathbf{n}}, \tag{2.9}$$

where $\tilde{\mathbf{h}}_k$ stands for the column vector $[\tilde{h}_{1k} \; \tilde{h}_{2k} \; \cdots \; \tilde{h}_{M_r k}]^T$, so that

$$\tilde{\mathbf{y}} = \tilde{\mathbf{h}}_1 \tilde{s}_1 + \tilde{\mathbf{h}}_2 \tilde{s}_2 \; \cdots + \tilde{\mathbf{h}}_{M_t} \tilde{s}_{M_t} + \tilde{\mathbf{n}}, \tag{2.10}$$

2.5.1 The Zero Forcing (ZF) Algorithm

The basic idea of zero forcing (ZF) is to pre-multiply $\tilde{\mathbf{y}}$ by a conditioning matrix $\tilde{\mathbf{G}}_{ZF}$ derived from the channel $\tilde{\mathbf{H}}$ that aims to reduce the inter-symbol interference (ISI) to zero for the current $\tilde{\mathbf{y}}$. Note that the presence of noise in $\tilde{\mathbf{y}}$ is ignored in the calculation of $\tilde{\mathbf{G}}_{ZF}$. The ZF algorithm contains the steps of interference nulling followed by slicing. The nulling step requires the calculation of a channel matrix inverse and the slicing step stands for the quantization operation from the nulled signal vector $\tilde{\mathbf{Z}}$ to the most likely symbol vector. With respect to Eq. (2.10), the ZF algorithm can be described as follows:

1. Nulling step:

$$\tilde{\mathbf{Z}} = \begin{bmatrix} \tilde{z}_1 \\ \vdots \\ \tilde{z}_{M_t} \end{bmatrix} = \tilde{\mathbf{G}}_{ZF} \tilde{\mathbf{y}} = \begin{bmatrix} \tilde{\mathbf{g}}_1 \\ \vdots \\ \tilde{\mathbf{g}}_{M_t} \end{bmatrix} \tilde{\mathbf{y}}, \tag{2.11}$$

The constructed matrix $\tilde{\mathbf{G}}_{ZF}$ in Eq. (2.11) meets the following constraints: $\tilde{\mathbf{g}}_1 \perp \tilde{\mathbf{h}}_k$ (where $k = 2, 3, \cdots, M_t$), so that $\tilde{\mathbf{g}}_1 \tilde{\mathbf{h}}_k = 0$, and also $\tilde{\mathbf{g}}_1 \tilde{\mathbf{h}}_1 = 1$. We

thus have $\tilde{z}_1 = \tilde{\mathbf{g}}_1\tilde{\mathbf{y}} = \tilde{\mathbf{g}}_1\tilde{\mathbf{h}}_1\tilde{s}_1 + \tilde{\mathbf{g}}_1\tilde{\mathbf{h}}_2\tilde{s}_2 \cdots + \tilde{\mathbf{g}}_1\tilde{\mathbf{h}}_{M_t}\tilde{s}_{M_t} + \tilde{\mathbf{g}}_1\tilde{\mathbf{n}} = \tilde{s}_1 + \tilde{\mathbf{g}}_1\tilde{\mathbf{n}}$. The other $\tilde{\mathbf{g}}_k$ vectors are computed similarly.

2. Slicing step: Apply a quantization operation (e.g., slicing) on $\tilde{\mathbf{Z}}$, appropriate for the 16-QAM modulation, to recover the corresponding symbol vector with the closest constellation point.

In the Zero Forcing algorithm, the conditioning matrix $\tilde{\mathbf{G}}_{ZF}$ (which is also called the ZF equalizer) is calculated as follows:

$$\tilde{\mathbf{G}}_{ZF} = \tilde{\mathbf{H}}^{\dagger} = (\tilde{\mathbf{H}}^{H}\tilde{\mathbf{H}})^{-1}\tilde{\mathbf{H}}^{H}, \tag{2.12}$$

where $(\cdot)^H$ is the conjugate transpose of a matrix. $\tilde{\mathbf{G}}_{ZF} = \tilde{\mathbf{H}}^{\dagger}$ is also known as the Moore-Penrose pseudo inverse [16] [17].

This definition of $\tilde{\mathbf{G}}_{ZF}$ ensures that the effects of the measured impairments are forced to zero (i.e., nulled) to totally remove the ISI, ignoring the possibility that some of the impairment is caused by noise. Thus, a noise-free environment is the ideal case for using the ZF algorithm. However in a normal noisy channel, the ZF algorithm's performance is limited because the effects of noise will tend to be amplified by multiplying the ZF equalizer to the received signal vector $\tilde{\mathbf{y}}$.

2.5.2 The Minimum Mean Square Error (MMSE) Algorithm

In the minimum mean square error (MMSE) algorithm, the basic strategy is similar to that of ZF. The difference is that a new inverse matrix $\tilde{\mathbf{G}}_{MMSE}$ is calculated to minimize signal distortion caused by both the channel $\tilde{\mathbf{H}}$ and the expected noise. The conditioning matrix is given by:

$$\tilde{\mathbf{G}}_{MMSE} = (\tilde{\mathbf{H}}^{H}\tilde{\mathbf{H}} + \frac{1}{\rho}\mathbf{I}_n)^{-1}\tilde{\mathbf{H}}^{H}, \tag{2.13}$$

where ρ denotes the SNR and $\mathbf{I}_n$ denotes the $n \times n$ Identity matrix. Note how the SNR ρ is considered in the calculation of the conditioning matrix $\tilde{\mathbf{G}}_{MMSE}$. The MMSE algorithm gives better performance than the ZF algorithm in the presence of additive Gaussian noise because it accounts for the average effects of the Gaussian noise while also minimizing the effects of ISI.

2.5.3 The Vertical BLAST (V-BLAST) Algorithm

The Bell Laboratories layered space-time (BLAST) detector was first proposed in [9]. It is an efficient MIMO detection algorithm that gives better BER performance than either ZF or MMSE at the cost of increased computational complexity. In V-BLAST, signal symbols are detected "vertically" from the same signal vector $\tilde{\mathbf{y}}$, that is, by detecting the symbol transmitted by each transmit antenna in turn in order of decreasing estimated SNR. V-BLAST achieves the better detection accuracy by exploiting interference cancellation. The principle of this algorithm is that the strongest (i.e., highest SNR) transmitted symbol is detected in the first step using either the ZF or MMSE criteria. Then the interference from this symbol on the M_r received MIMO signals is predicted and subtracted away to eliminate the interference of the symbol from the M_r signals. The same steps are repeated to detect the remaining transmitted symbols. In this way, we can cancel the interference caused by previously detected symbols to offer more accurate detection for the next detected symbol. But we also need to pay an increasing calculation cost when the number of antennas grows. The V-BLAST algorithm is more expensive computationally than ZF and MMSE, but the cost still grows linearly in the number of antennas. The algorithm is as follows:

1. Initialization: $i = 1$. Compute the first conditioning matrix $\tilde{\mathbf{H}}^{\dagger}$ from $\tilde{\mathbf{H}}$.

2. Ordering: Set the i-th conditioning matrix $\tilde{\mathbf{G}}_i = \tilde{\mathbf{H}}^{\dagger}$. Calculate the smallest norm value over all columns of $\tilde{\mathbf{G}}_i$, $k_i = \arg\min_j \left\|(\tilde{\mathbf{G}}_i)_j\right\|^2$, where k_i denotes the index of the column with the minimum norm. Select this column, $\tilde{\mathbf{g}}_{k_i} = (\tilde{\mathbf{G}}_i)_{k_i}$. Essentially, we are ordering the undetected symbols in decreasing order of expected post-detection SNR.

3. Nulling and Slicing: $\hat{\tilde{s}}_{k_i} = quantize(\tilde{\mathbf{g}}_{k_i}\tilde{\mathbf{y}}_i)$. $\tilde{\mathbf{y}}_i$ is the received signal vector element in Eq. (2.5). Null the interference on symbol k_i from the other $M_t - i$ undetected symbols. Then slice $\tilde{\mathbf{g}}_{k_i}\tilde{\mathbf{y}}_i$ to detect $\hat{\tilde{s}}_{k_i}$.

4. Interference Cancellation: Compute $\tilde{\mathbf{y}}_{i+1} = \tilde{\mathbf{y}}_i - \tilde{\mathbf{h}}_{k_i}\hat{\tilde{s}}_{k_i}$. Then remove the k_i-th column $\tilde{\mathbf{h}}_{k_i}$ from the channel $\tilde{\mathbf{H}}$ to reflect the fact that the effects of one

transmit antenna can now be removed. So we predict the interference caused by the detected symbol, and then subtract this interference from all of the MIMO signals.

5. $i = i + 1$, go back to step 2 until all the symbols are detected (i.e., until $i > M_t$).

2.5.4 Performance of the Linear Algorithms

In order to compare the performance of these different linear algorithms, we consider a 4×4 MIMO system with 16-QAM, which is a commonly studied configuration in the research literature.

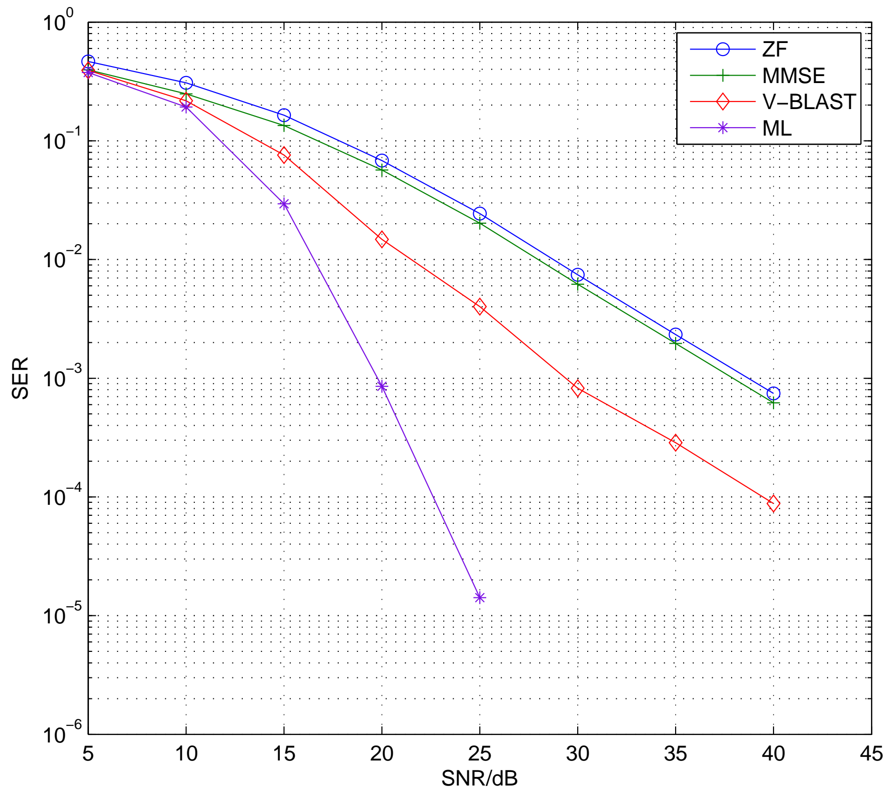

Figure 2.3: Performance of three linear MIMO detection algorithms (ZF, MMSE and V-BLAST) compared to the optimal detection method (ML detection) for $M_t = M_r = 4$, and 16-QAM modulation. Each data point represents at least 100 detection errors.

Fig. 2.3 shows the big performance gap between the optimal and suboptimal

detection algorithms according to the SER vs. SNR characteristic. The three linear symbol detection algorithms have relatively low computational complexity but their SER performance is relatively poor at high SNRs. The SER performance of V-BLAST is limited by error propagation effects of symbol detection errors for the first detected symbols [18].

2.6 Overview of the Sphere Detection Algorithm

All three of the linear MIMO detection algorithms, which we described above, provide suboptimal detection. Optimal detection is guaranteed by the ML algorithm, which is expensive computationally. Clearly there is a trade-off between detection accuracy and detector complexity. Consequently, researchers have sought algorithms with near-optimal performance but with lower complexity than ML detection. The sphere detector (SD) algorithm has proven to be such a method. The real-valued system expressed as Eq. (2.6) and Eq. (2.7) is used in this thesis for the SD algorithms, following standard practice.

The basic idea of SD is to find the closest point within the lattice Ω of possible symbol vectors that lie within a certain hypersphere of radius d centred on a symbol vector estimated using a linear detector such as MMSE. It is convenient to factor the channel matrix $\mathbf{H} = \mathbf{QR}$ using QR decomposition [19], where $\mathbf{Q}$ is an unitary matrix, i.e. $\mathbf{Q}^H\mathbf{Q} = I$, and $\mathbf{R}$ is an upper triangular matrix with non-negative diagonal elements. Let $\mathbf{z} = \mathbf{Q}^H\mathbf{y}$, then according to Eq. (2.6),

$$\mathbf{z} = \mathbf{Q}^H(\mathbf{QRs} + \mathbf{n}) = \mathbf{Q}^H\mathbf{QRs} + \mathbf{Q}^H\mathbf{n} = \mathbf{Rs} + \mathbf{Q}^H\mathbf{n} \tag{2.14}$$

Thus, Eq. (2.8) can be rewritten as follows:

$$\hat{\mathbf{s}} = \arg\min_{\mathbf{s}\in\Omega} \|\mathbf{z} - \mathbf{Rs}\|^2 = \arg\min_{\mathbf{s}\in\Omega} \sum_{i=1}^{n} |z_i - \sum_{j=i}^{n} r_{ij}s_j|^2, \tag{2.15}$$

where $n = 2M_t$ represents the dimension of $\mathbf{H}$, and r_{ij} are elements of the $\mathbf{R}$. Then, the partial Euclidean distance (PED) after detecting symbol values $s_n, s_{n-1}, \ldots, s_k$ in symbol vector positions $n, n-1, \ldots, k$, where $n \geqslant k \geqslant 1$, can be written as follows:

$$T_k = \sum_{i=k}^{n} |z_i - \sum_{j=i}^{n} r_{ij}s_j|^2 \leqslant d^2 \tag{2.16}$$

If $T_k > d^2$ for a symbol s_j, all fully detected symbol vectors based on the given partially detected symbol vector will be pruned away and discarded. In this way, the complexity of the sphere detection algorithm will be reduced compared with the exhaustive ML detection.

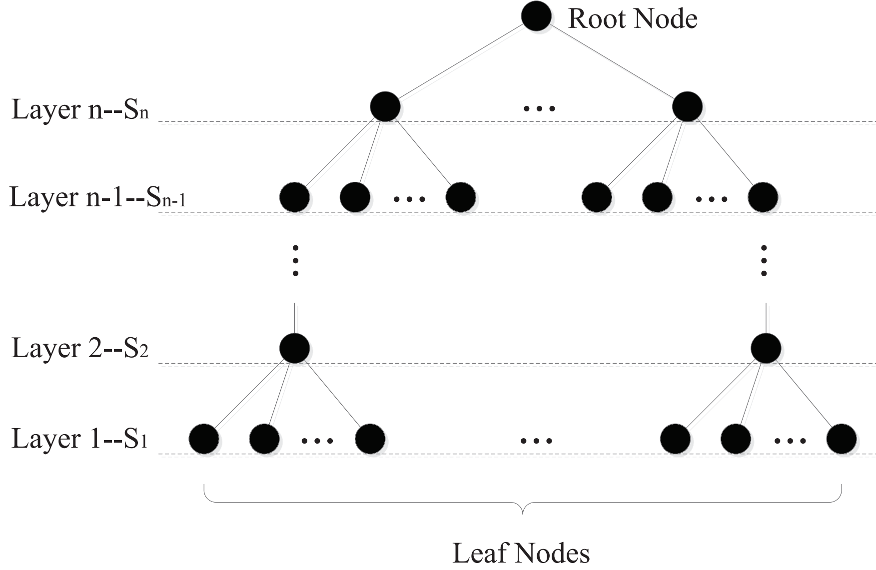

Figure 2.4: Search tree model for successive symbol detection

it is common to convert the sphere detection algorithm into a tree search problem. Fig. 2.4 shows the model of the search tree we applied in this thesis. The root node at the top of the tree corresponds to the start of the search for the best symbol vector $\hat{\mathbf{s}}$. The leaves of the tree at the bottom correspond to the set of fully detected candidate symbol vectors. The tree has n ($n = 2M_t + 1$) layers including the root node and each traversed node has M_c sub-nodes under it so that the total number of nodes in this tree will be $M_c \times (M_c^0 + M_c^1 + \cdots + M_c^{n-1})$. The search starts at the root node before the first symbol has been detected. As the search progresses from the root node, symbol selections are made going from the n-th layer to the $(n-1)$-th layer, etc. on down to the 1-st layer. Then the least-cost path from the root node down to a leaf node is the detected received signal vector. Following conventional tree search theory given a general graph theory reference [20], we distinguish between two basic kinds of sphere detection algorithms:

- The Depth-First Tree Search SD algorithms include the Fincke-Pohst (FP)

algorithm and Schnorr-Euchner (SE) enumeration. The relative complexity of these two methods varies with the system's SNR.

- The Breadth-First Tree Search SD algorithms include the K-Best algorithm with the fixed complexity.

2.6.1 The Fincke-Pohst (FP) Sphere Detection Algorithm

The details of the FP sphere detection (FP-SD) algorithm are given in [21]. One of the key ideas is that the initial radius d is defined as

$$d^2 = \alpha n \sigma^2, \tag{2.17}$$

where σ^2 is the variance of the noise vector **n**. The probability $1-\epsilon$ that a sphere of radius d will enclose the correct signal vector is given by:

$$\int_0^{\frac{\alpha n}{2}} \frac{\lambda^{\frac{n}{2}-1}}{\Gamma(\frac{n}{2})} e^{-\lambda} \mathrm{d}\lambda = 1 - \epsilon \tag{2.18}$$

which is the cumulative density function of a χ^2 random variable with n degrees freedom. The initial search radius $d = \sigma\sqrt{\alpha n}$ should be made large enough by adjusting α to make sure the received signal vector can be found with high probability. Thus, $1-\epsilon$ should be a value that is close to 1 (i.e., $\epsilon = 0.01$). Then we can re-write Eq. (2.16) to derive the upper and lower bounds for s_i if the i-th level,

$$\left\lceil \frac{-d_i + z_i^{,}}{r_{ii}} \right\rceil \leqslant s_i \leqslant \left\lfloor \frac{d_i + z_i^{,}}{r_{ii}} \right\rfloor, \tag{2.19}$$

where $z_i^{,} = z_i - \sum\limits_{j=i+1}^{n} r_{ij} s_j$.

The FP algorithm is as follows:

Inputs: n, **R**, **z**, d

1. Initialization: $i = n$, $d_i = d$, and the PED which was defined in Eq. (2.16) is set initially to be infinite. PED is updated later with the lowest PED found so far. First point is searched from the first constellation (ω) point.

2. Set $upperbound = \left\lfloor \frac{d_i + z_i^{,}}{r_{ii}} \right\rfloor$ and $lowerbound = \left\lceil \frac{-d_i + z_i^{,}}{r_{ii}} \right\rceil$ according to Eq. (2.19).

3. Determine the number of nodes within the bound. If there's the node within the bound, go to step 5; else go to step 4.

4. $i = i+1$. If $i = n+1$, terminate the algorithm and return the detected symbol vectors and go to step 7; else go to step 3.

5. If $i = 1$, go to step 6; else $i = i - 1$, and update the search radius $d_i = d_{i+1} - (z_{i+1} - \sum_{j=i+1}^{n} r_{i+1,j} s_j)^2$ and go back to step 2.

6. The last level has been reached. Calculate the PED of this detected symbol vector **s** as $ped = \|\mathbf{y} - \mathbf{Hs}\|^2$. Compare this ped with the previous lowest PED. If $ped \leqslant$ PED, save **s** and assign PED $= ped$, then go to step 3.

7. If the returned symbol vector **s** is empty, reduce the ϵ to get a larger radius d. Restart the algorithm from step 1.

Both the forward (going down layers) and backward (going up layers) tree search are applied in a depth-first search order, so that the performance of the FP-SD algorithm approaches that of ML detection, However, the cost in computational complexity is extremely high, especially when the size M_c of the QAM constellation and the number M_t of transmit antennas increase.

2.6.2 Schnorr-Euchner (SE) Enumeration

In the SE sphere detection (SE-SD) algorithm [22], [23], we begin the search from the Babai point (BP) s_i, which is the Zero-Forcing solution at the $n-th$ level. We then use Eq. (2.20) to define a zigzag search path to determine the next node. As the search proceeds, we keep the **s** which has the smallest PED encountered so far.

$$s_i = s_i, s_i - 1, s_i + 1, s_i - 2, s_i + 2, \ldots \tag{2.20}$$

The initial radius d of this algorithm is generally set to be infinite. However, in this algorithm, it is set to $2^{1}0$ due to the finite constellation used, to avoid an infinite loop [23].

The SE Enumeration strategy:

Inputs: n, **R**, **z**

1. Initialization: $i = n$, $bestdist = 2^10$, the initial PED $dist_i = 0$, $s_i = quantize(z_i)$ is the BP based on the constellation points set, and error $e = z_i - r_{ii}s_i$. Record the sign of e: $step_i = sign(e)$ to determine next direction of enumeration.

2. $newdist = dist_i + e^2$. If $newdist < bestdist$, go to step 3; else go to step 6.

3. If $i > 1$, $i = i - 1$, go to step 4; otherwise the lowest level has been reached and so save **s** as the detected symbol vector and update $bestdist = newdist$. Go to step 5.

4. $tempZ_i = z_i - \sum_{j=i+1}^{n} r_{ij}s_j$, new PED $dist_i = newdist$, $s_i = quantize(tempZ_i)$ based on the constellation points set, $e = tempZ_i - r_{ii}s_i$. Record the sign of the error e, $step_i = sign(e)$. Go to step 2.

5. $i = i + 1$, $e = 2^5$ (2^5 ensure the finite loop when it go back to the step 2, suggested in [23]) to make sure that the lower level will be discarded because the condition is unsatisfied. Start the loop for $n = 1 : 2$. Enumerate from the BP s_i according to Eq. (2.20). If the next s_i is within the constellation, then break the loop and go back to step 2; otherwise, continue the loop and keep searching within BP, then go to step 2.

6. if $i = n$, terminate the algorithm and return the detected symbol vector **s**; else go to step 5.

As can be seen from Fig. 2.5, the SER performance of SE-SD can closely approach that of exhaustive ML detection. Note that, because the Zero-Forcing solution ensures that the start of the search will be closer to the optimal point compared to FP-SD (in FP-SD, the search starts from the first point of the M_c constellation), the complexity of SE-SD is much lower than FP-SD even though it still takes a long time to find the optimal ML solution when the SNR is low.

2.6.3 The K-Best Sphere Detection Algorithm

As mentioned above, the K-Best sphere detection (K-Best SD) algorithm uses a breath-first search strategy. Starting from the n-th level, we keep the K nodes that

have the smallest PEDs at each level to obtain a matrix that comprises K detected vectors $\mathbf{s}$. We then pick the symbol vector $\mathbf{s}$ with the smallest PED as the output result after the tree search is finished.

The basic K-Best algorithm is as follows:

Inputs: n (number of levels), K (retained nodes per level), $\mathbf{R}$, $\mathbf{z}$ (these two matrices are the result of QR decomposition and are used to calculate the PEDs)

1. Initialization: $i = n$, the initial detected symbol vector $\mathbf{s}$ is set to be empty. Calculate the PEDs of each node within the M_c constellation points according to Eq. (2.16). Pick the K partial symbol vectors with the smallest PEDs.

2. $i = i - 1$ and begin searching the next level down.

3. Extend the surviving partial symbol vectors and obtain $M_c K$ contender paths. Select the K partial symbol vectors with the smallest PEDs and update the path history with them.

4. If $i = 1$, terminate the algorithm and return the symbol vector $\mathbf{s}$ that has the smallest PED; otherwise, return to step 2.

If K is large enough, which means the surviving paths contain most if not all the closest symbol vectors, the performance of K-Best SD algorithm approaches that of exhaustive ML detection [24]. However, in the K-Best algorithm, the complexity is proportional to the number $M_c K$ of searched paths at each level (expecting the n-th level with M_c paths), so it will increase linearly with increasing K.

2.6.4 Pre-processing the Channel Matrix

During the processing of the sphere detection algorithms above, it is clear that the quality of the estimate of the channel $\mathbf{H}$ will influence both the search complexity and the performance. In other words, when the channel's SNR is high enough, it should be much easier for these algorithms to correctly detect the symbol vector. In addition, preprocessing the channel before detection might produce better performance.

As with ZF and MMSE, it is common to condition the signal vector by pre-multiplying by the Moore-Penrose pseudo-inverse (denoted as $(\cdot)^{\dagger}$) which is computed from the channel matrix **H** as Eq. (2.12) in real valued system:

$$\mathbf{G} = \mathbf{H}^{\dagger} = (\mathbf{H}^{H}\mathbf{H})^{-1}\mathbf{H}^{H} \tag{2.21}$$

Here we choose the ZF equalizer instead of MMSE equalizer because as has been mentioned in the sections of ZF and MMSE algorithms, the ZF equalizer can totally eliminate inter-symbol interference if the noise is negligible. The purpose of the ZF equalizer during preprocessing is to order the channel from the strongest layer to the weakest layer which has nothing to do with the noise signal. Through this pseudo-inverse, we can accomplish effective interference cancellation prior to symbol detection. The conditioned signal vector $\hat{\mathbf{s}}$ is given by:

$$\hat{\mathbf{s}} = \mathbf{G}\mathbf{y} = \mathbf{s} + (\mathbf{H}^{H}\mathbf{H})^{-1}\mathbf{H}^{H}\mathbf{n} \tag{2.22}$$

To achieve the smallest detection error on the $i-th$ layer, the row $\mathbf{g}_i$ of **G** should have the minimum Euclidean norm value, to minimize the interference noise from the other undetected symbols. According to this, we should sort the rows of channel matrix **H** to obtain better performance. The preprocessing algorithm is as follows:

Inputs: n, **H**, **y**

1. Initialization: $i = n$, $\mathbf{h} = \mathbf{H}$, $\mathbf{p} = (1, 2, \ldots, n)$ which denotes the reordered subscripts of **H**.

2. Start the loop from $i = n$, $\mathbf{G}_i = \mathbf{H}^{\dagger}$ according to Eq. (2.21).

3. Calculate the minimum norm among $\mathbf{g}_i, \ldots, \mathbf{g}_n$: $k_i = \arg\min_{j=1,\ldots,i} \|\mathbf{g}_j\|^2$, and exchange the columns i and k_i in **H**, and update **p**.

4. Repeat step 2 and step 3 with $i = i - 1$ until $i = 1$. then

5. If $i = 1$, the loop is finished. Calculate the **Q**, **R** from the new reordered **H** using QR decomposition. Return the **Q**, **R**, **H** and the corresponding ordered subscript vector **p**.

After the preprocessing, the resulting new **Q**, **R**, **H** can be used in the sphere detection algorithm and the algorithms' complexity can be partially reduced which can also be shown in Fig. 2.7. One thing to note here is that the symbols in the detected vector **s** should be reordered relatively according the ordered subscript **p**.

2.6.5 Performance of the Sphere Detection Algorithms

The operation environment of the system is the same as the one that was assumed for the linear detection algorithms. The plots in Fig. 2.5 show that the sphere detection algorithms achieve much higher detection accuracy than the suboptimal, algorithms illustrated in Fig. 2.3, while costing much more in computation.

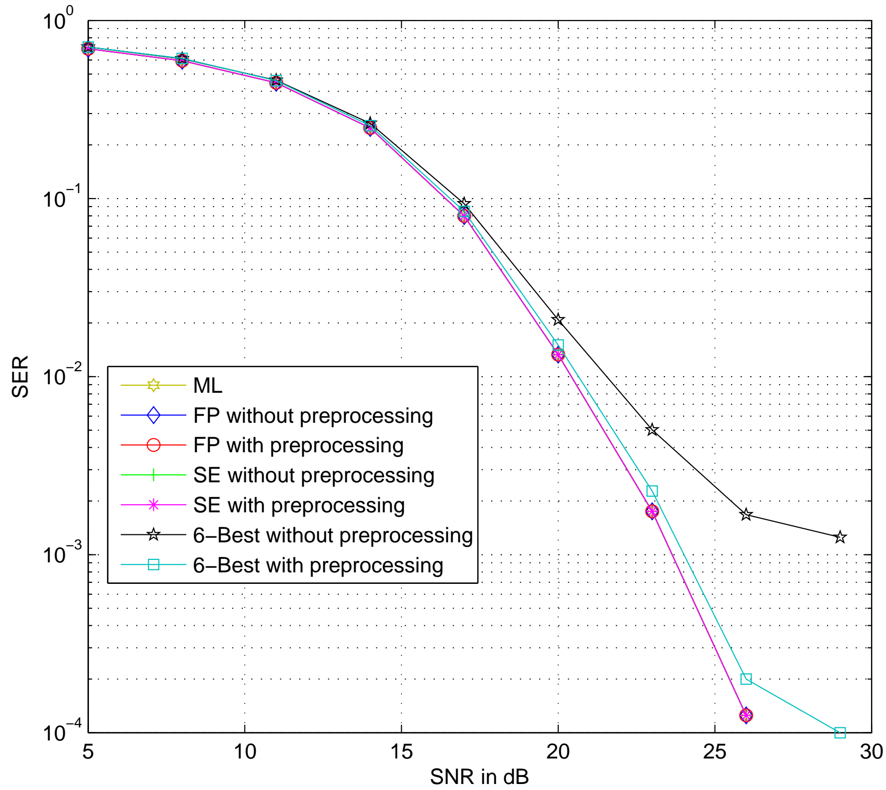

Figure 2.5: Performance of three detection algorithms (FP, SE, K-Best when $K = 6$) with and without preprocessing and for a $M_t = M_r = 4$, 16-QAM MIMO system. Each data point represents at least 100 detection errors.

Fig. 2.5 shows the SER v.s. SNR performance of the different sphere detection algorithms with preprocessing. It shows that the FP and SE algorithms approach

optimal ML detection performance while the K-Best ($K = 6$)'s performance approaches optimal performance only after applying the preprocessing method.

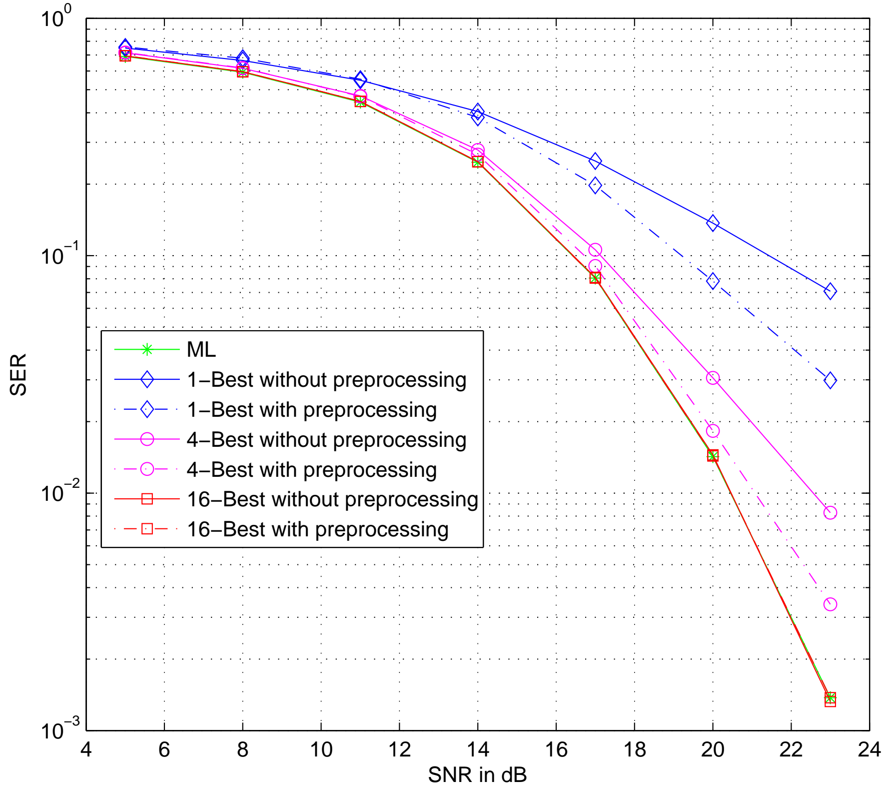

Figure 2.6: Effects of preprocessing for a $M_t = M_r = 4$, 16-QAM MIMO system when $K = 1, 4$, and 16. Each data point represents at least 100 detection errors.

The advantage of employing preprocessing can be observed much more clearly in Fig. 2.6, where we observe the benefits of preprocessing for K-Best search for $K = 1, 4$, and 16. As described in the algorithm, the performance of the K-Best algorithm improves with increasing K. Note that when the number K of selected nodes equals one, the algorithm performs similarly to ZF detection at the opposite extreme. When K is large enough to contain all the expanded nodes with very high probability, the performance matches that of exhaustive ML detection.

In Fig. 2.7, we compare the complexity cost, which are proportional to the total number of operations required by these algorithms. To plot this figure, we set a variable to accumulate the number of nodes that have been traversed by each of the

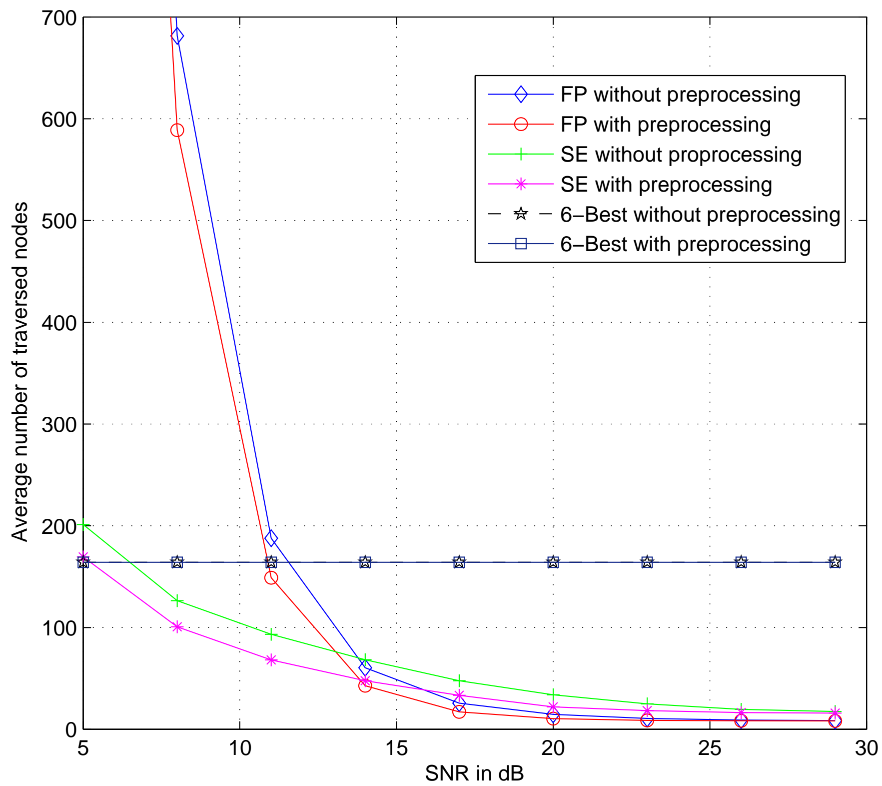

Figure 2.7: Complexity of three detection algorithms (FP, SE, K-Best when $K = 6$) for a $M_t = M_r = 4$, 16-QAM MIMO system.

detection algorithms. We can see in Fig. 2.7 that, in a poor low-SNR environment, the complexity of the FP sphere detection algorithm is much higher than that of the other two algorithms. However, when the SNR is greater, it becomes easier to detect the correct symbol vector in both the FP and SE algorithms. For the K-Best algorithm, the complexity stays fixed as expected. However, the complexity of the K-Best algorithm increases several-fold when the K becomes larger because at each level, the computation amount of each selected node depends on the size M_c of the constellation.

$\sim$

Chapter 3

Parallelism and the Graphics Processing Unit

3.1 Parallelism

Traditionally, programs are written to produce serial data manipulations and calculations. The execution time of a calculation is directly proportional to the required number of representative CPU operations. For cases where we need to deal with a large amount of data, the data storage capacity is also a limitation if only one processor is considered. To solve these problems, parallel processing on parallel hardware is one strategy that can be applied to speed up the processing. In parallel processing, the problem is divided into several sub-programs that are executed at the same time on different processors so that the total processing time is reduced. The shrinking size of semiconductor transistors and wires is allowing more and more processing cores to be provided on each chip, so parallel hardware is now widely available and relatively inexpensive.

3.1.1 Classification of Parallelism

There are two major ways of implementing hardware parallelism, pipelining and multiprocessing.

Pipelining

In a classical Von Neumann computer architecture, binary data and program instructions are stored in a shared memory [3]. A single central processing unit (CPU)

fetches instructions and executes them one by one.

Various strategies have been employed to speed up calculations.

- Use faster technologies to design CPUs with faster clocks.
- Use a cache memory hierarchy to speed up the average time for memory accesses.
- Use branch prediction to allow instructions to be prefetched from main memory into cache and thus speed up the average time for instruction fetches.
- Hardware parallelism of various kinds.

An instruction pipeline is used to improve the processing efficiency of one processor. Since the processor is driven by the system clock, the fetching, decoding and execution of each instruction is divided into several steps by clocked registers. The operations between the registers occur in parallel, keeping the corresponding functional blocks busier and speeding up the instruction throughput. When the second step of an instruction is executed, the first step of the next instruction can also be fetched by the processor at the same time. Branching can cause lost time in a pipelined system because intermediate data results in the pipeline become invalid. Branch prediction attempts to minimize this inefficiency by minimizing the probability of mispredicted branches.

Multiprocessing

While the pipelines architecture is applied within one processor, the multiprocessing approach to parallelism uses multiple processors.

There are four major kinds parallelism according to Flynn's taxonomy [25] [26] [27]. This taxonomy represents theoretical extremes of computer architecture. Real computer architectures incorporate different kinds of parallelism at different levels of their architecture.

1. Single Instruction, Single Data (SISD)
    - Non-parallel classical Von Neumann architecture.

- The instructions are fetched by the one CPU from the common memory and executed one-by-one at a rate determined by the system clock. The execution time per instruction is determined by the clock period, and the average number of clock cycles per instruction.
- A single data stream is processed serially by one CPU.

2. Single Instruction, Multiple Data (SIMD)

    - Parallel structure is present in the data memory and in the data processing hardware. In other words, there are multiple parallel data paths.
    - The same instruction stream is executed on the parallel data streams within the parallel data processing hardware. There is a shared instruction fetch and decode unit that broadcasts the shared control signals to the parallel data paths.
    - This form of parallelism is efficiently applied in graphics/images processing. Graphical data processing operations typically involve non-interacting local neighbours of pixels, so these operations can proceed in parallel in the parallel data paths.

3. Multiple Instruction, Single Data (MISD)

    - Parallel structure is present in the CPU but not in the data memory or the data processing hardware.
    - Multiple instruction streams are executed simultaneously by multiple processing units.
    - Special partitioned memory hardware and/or algorithmic constraints must be used to avoid conflicting multiple write operations to the same data memory locations.
    - The usage of the MISD is not as widespread as SIMD. One of the few examples [27] is the experimental Carnegie-Mellon C.mmp computer (1971) [28] [29].

4. Multiple Instruction, Multiple Data (MIMD)

- The most flexible form of parallel structure, with parallelism in both the CPU and the data memory.
- Totally independent instruction streams are executed in parallel by different processors.
- Each data stream is processed by a different processor using a separate instruction sequence.
- Modern multi-core personal computers can apply this strategy to get the speed-up during data processing if the calculation can be partitioned into independent or loosely interacting parallel threads.

The SISD architecture is the traditional computer model when algorithms are developed. However, there are disadvantages to the SISD architecture. Many problems have inherent parallelism that could be exploited for faster execution on parallel hardware. The simplest form of parallelism (SIMD) involves performing the same instructions with different data on different processors. The most complex form of parallelism (MIMD) is to execute different commands with different data. In this thesis, we investigate how each of these two forms of parallelism can reduce the execution time of MIMO detection algorithms in the communications area.

3.1.2 The Limits of Parallelism - Amdahl's Law

Ideally, the acceleration of the parallelism should increase linearly with the number of parallel processors that are applied. However, in most problems, not all of the commands in the algorithm can be executed in parallel so that the achievable speed-up of a parallel program with multi-processors is limited by the inherently serial part of the algorithm. Amdahl's law [30] gives the potential speed-up of a program with serial and parallel parts. Amdahl's law is given by:

$$S(n) = \frac{1}{(1-P) + \frac{P}{n}}, \tag{3.1}$$

where $S(n)$ is the speed-up factor of the parallelism with n multiprocessors, P stands for the proportion of the program that can be executed in parallel, and $(1-P)$ represents the serial proportion. Notice that, when the number n of parallel paths

tends to infinity, the maximum possible speed-up is limited by the non-parallelized portion of the program, no matter how large the degree n of parallelism.

The Amdahl's law is illustrated in Fig. 3.1:

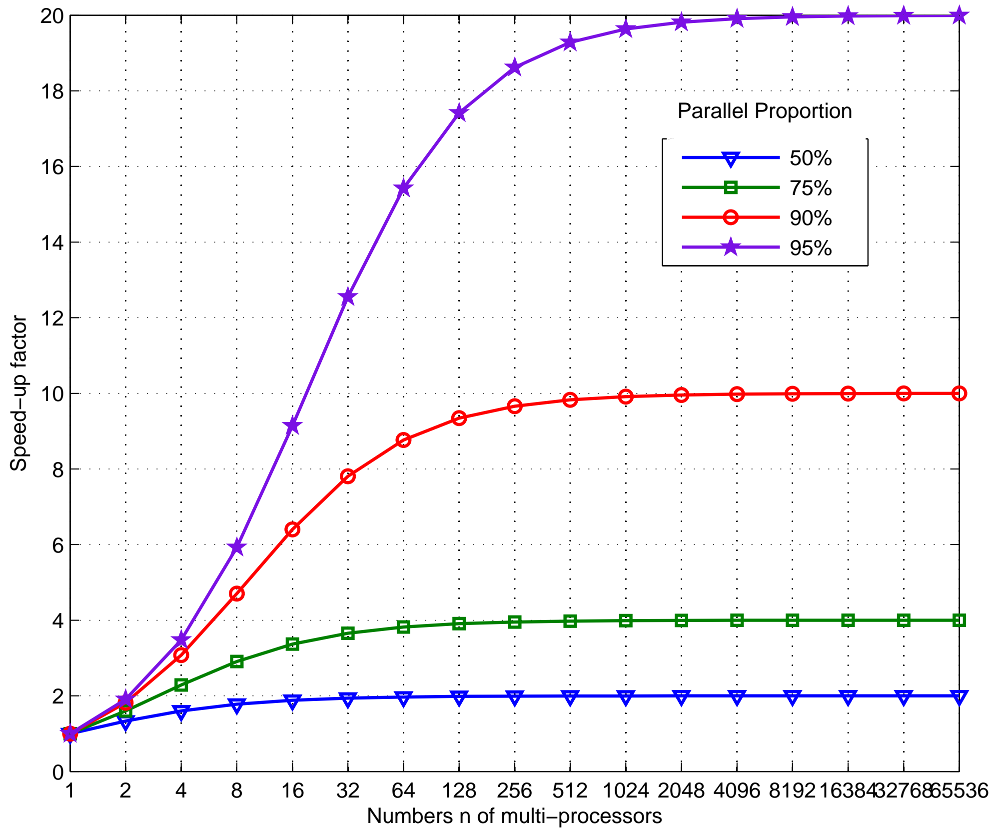

Figure 3.1: Amdahl's Law. The speed-up of a program executed in parallel by different numbers n of multiprocessors with different degrees of parallelism.

Because of this limitation, we must be careful to pick a suitable number of parallel paths and also convert the highest proportion of the algorithm into parallel form as possible to achieve the most benefits from the available parallelism.

3.2 The Graphics Processing Unit

Graphics data processing is an economically important but computationally demanding class of problems faced by modern computers. The graphics processing unit (GPU) is a subsystem in modern computers that is provided with extensive

hardware parallelism that is used to speed up graphical data processing. GPUs are widely used in consumer PCs, supercomputers, game consoles and even cell phones.

3.2.1 Architecture of the GPU

The concept of the GPU was proposed in 1999 [31] and applied initially to the personal computer. The company NVIDIA released "the world's first" GPU, the GeForce 256, with the ability to "process a minimum of 10 million polygons per second" [31]. From then on, GPU technology evolved rapidly. General-purpose computing on graphics processing units (GPGPU) is a new trend that attempts to use the parallel computing power of the GPU for a wider range of programming problems, beyond graphics processing.

General Structure of GPU

A GPU contains an array of processing cores, distributed memories and global memories interconnected with high bandwidth buses. Typically, the same instruction is executed by the multiple GPU cores on parallel streams of data.

The memory transfer between GPU and CPU is shown in Fig. 3.2. The global memory, constant cache and texture cache are shared among the multiprocessors. The constant and texture cache are read-only and can be accessed faster than shared memory. The global memory supports data read and write operations and is accessed by all GPU threads and the host.

GPU with CUDA Cores

CUDA is short for "Compute Unified Device Architecture" which was provided as a software development environment by NVIDIA to support GPGPU computing on their GPUs. This environment includes new features to support general-purpose computation. Parallel code is executed by different CUDA threads running on multiple parallel CUDA cores. All the threads in one multiprocessor are independent of each other but execute the same instructions following the SIMD model. The SIMD model imposes a strict form on the parallel computation.

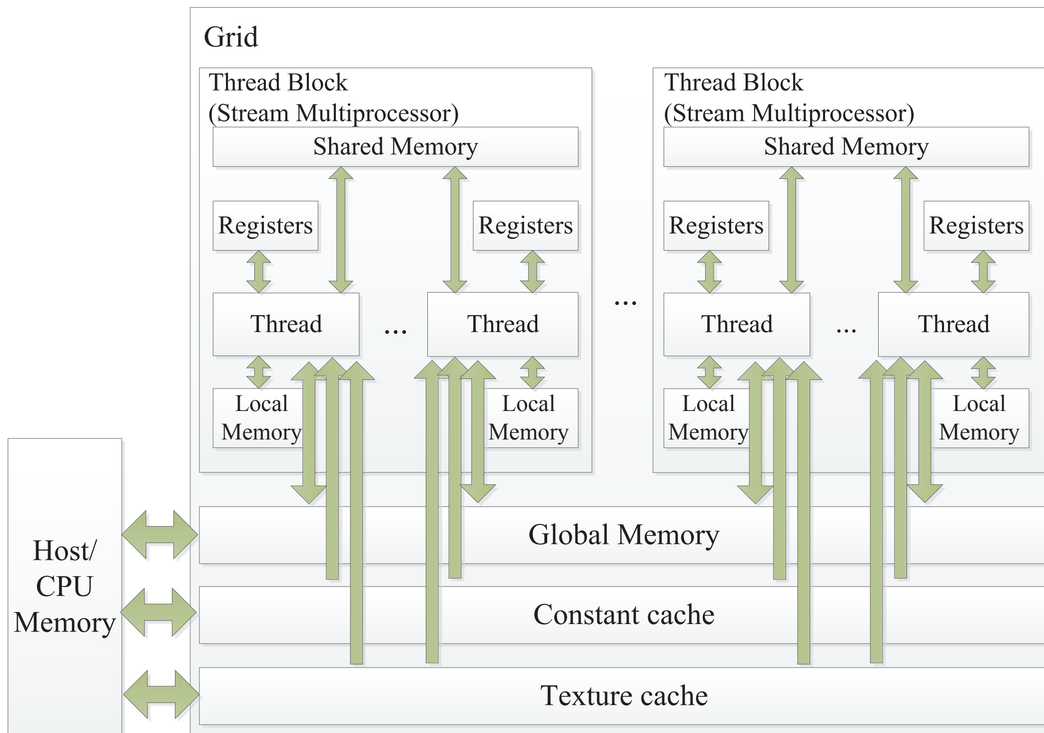

Figure 3.2: Memory structure of a GPU [1]

The recent third generation stream multiprocessor (SM) introduced by NVIDIA brings more innovations in the architecture [1]. Each SM includes 32 CUDA cores, where each core includes a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). This architecture also complies with the IEEE 754-2008 floating point standard. A MIMD architecture is used within SMs.

Important GPU-related terms are defined below.

- Threads: A smallest unit of program execution in parallel processing. Each thread typically has its own local memory for storing local variables.
- Warp: A set of threads which are running in parallel at the same time. A warp consists of up to 32 threads. The concept of warp was introduced in CUDA by NVIDIA.
- Thread Block: A group of threads are organized into a thread block, and also a block can be made up of warps. These threads share memory space and cooperate with each other via barrier synchronization.

- Grid: An array of thread blocks that execute the same parallel program and that access the global memory. These blocks are executed one by one, so that the synchronization does not exist among these blocks.

- CUDA Stream: A host initiated sequence which contains a list of grids executing in order.

The concepts described above are illustrated in Fig. 3.2.

3.2.2 The GPU Programming Model

In order to make effective use of GPUs, several programming models have been proposed:

CUDA C

CUDA refers both to a general-purpose parallel platform and a programming environment. NVIDIA designed a new programming language, CUDA C, with a compiler for GPGPU computing. CUDA is in fact an extension to the general-purpose C language. It allows instructions, which are called “kernels”, to be executed by different CUDA threads in parallel instead of following the serial operation of regular C functions. Since CUDA C was released by NVIDIA, the structure of this language fully follows the GPU-related terms as described above.

OpenCL

Open computing language (OpenCL) is a vendor-independent environment for parallel programming. It is intended to support heterogeneous parallel computing architectures that include different combinations of CPUs, GPUs, DSPs and so on. OpenCL is more portable than CUDA C because CUDA C only supports the NVIDIA hardware platform. OpenCL language is in fact based on C with extensions that support auto-configuration that adjusts automatically to the available hardware and parallel programming constructs. An OpenCL program is divided into two parts: the host part and the device part. The host part, which refers to the CPU (the host in general), provides an Application Program Interface (API) to

manage the programming on the device parts, to allocate memory resources and to control the run-time environment. The device parts, which could include GPUs and DSPs, are responsible for parallel functions offloaded from the CPUs.

In OpenCL, the task in the device part is divided into work groups which refer to CUDA thread blocks. All these work groups are organized by ND range (next organization level). A work group organizes all the work items that correspond to the CUDA threads within it. At the host side, all the instructions follow the SIMT model, which stands for Single Instruction Multiple Thread, which means that the same instruction is executed on the different threads at the same time.

Jacket

Jacket, which was marketed from 2007 to 2012 by AccelerEyes (Atlanta, GA), is another parallel GPGPU computing platform. Jacket is designed to accelerate MATLAB-based codes running on GPU-equipped PCs that have CUDA technology on the GPU. Jacket provides parallel extensions of data types and functions for MATLAB. Most of the Jacket commands look as same as the original MATLAB codes, but with several limitations governing their usage. MATLAB is a widely used technical programming language and environment for many different kinds of fields in both academic and industry areas, such as signal processing, data analysis, mathematical computations, image processing, and application development. Jacket extends MATLAB to make the GPU data structures and operations much more visual and easier to be understand, and to make sure that the GPU applications can work properly in MATLAB environment. In this thesis, the algorithms were originally written in MATLAB, so our initial GPU acceleration strategy was to exploit Jacket.

In Jacket programming, data can be either moved (i.e., cast) cast from the CPU memory to GPU memory or created on the GPU's own memory, depending on the functions that are used. According to the Jacket documentation [32], it costs significant time to transfer data between the GPU and CPU and that bottleneck reduces the benefits from GPU acceleration. Thus, as much as possible, it is better to create the data on GPU directly and then cast the final result to the CPU in a final

Code Listing 3.1: Simple Example to Generate and Casting Random numbers on/to the GPU using Jacket Library in MATLAB

```
% Casting a matrix on the GPU
a = randn(N); % N is the size of matrix
b = gdouble(a); % matrix b is a parallel data structure on the GPU

% Creating a matrix on the GPU
A = gzeros(N,N,Parallelism); % N is the size of matrix
B = grand(N,N,Parallelism);     % Parallelism is the degree of
    parallelism
C = A+B; % matrices A, B and C are all parallel structure on the
    GPU
```

phase to collect and possibly plot the final results.

Here are some of the Jacket functions for creating parallel data structures that reside on the GPU [32].

- gsingle, gdouble, glogical, gint8, gunit8, gint32, guint32: These functions cast data structures from the CPU to GPU memory.

- gzeros, gones, geye, grand: These functions create a matrix of zeros, ones, the identity matrix, random matrix directly in the parallel GPU cores.

Code Listing 3.1 shows a simple example that generates parallel data on the GPU. All of these GPU data structures are manipulated by parallel operations on the GPU. The last input argument is usually used to specify the number of parallel GPU cores to be used.

Many parallel extensions of basic operations are supported on the GPU [32], such as matrix and array's arithmetic operations, relational operations, logical operations, diagonal matrices and diagonals of matrix (diag), LU matrix factorization (lu), orthogonal-triangular decomposition (qr), sorting array elements in ascending or descending order (sort), etc.

Parallelism can be performed in a loop-like control structure. Instead of launching each of the loops sequentially, as in the original MATLAB for-loop, Jacket uses the gfor-loop to vectorize it on volumes as well so that the original loop iterations be performed simultaneously on parallel GPU cores. The iterator of the gfor-loop controls the degree of parallelism.

It is often possible to avoid using parallelism that is explicitly specified using a gfor-loop, and to instead rely on the implied use of parallel operations on parallel variables. Such implicit vectorization usually provides better performance than the explicit parallelism using the gfor construct. For example, use $a = b + c$ instead of looping $a(ii) = b(ii) + c(ii)$ in a gfor loop with $ii = 1 : parallelism$.

There are many built-in functions that are supported for parallel operation within a gfor-loop such as fft, sum, max, min, ect. However, these functions have restrictions that we must consider [32]. Here are some of these key constraints.

- All iterations within one gfor-loop must be independent of each other. Data dependencies are not allowed among different iterations of the gfor-loop.

- Conditional statements are not allowed inside a gfor-loop. Conditional execution can be implemented in many cases by multiplying by a boolean condition. Fig. 3.3 shows this way of avoiding conditional statements. where

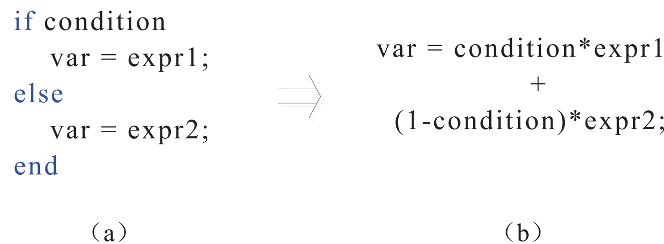

Figure 3.3: If-then-else construct replaced with a multiplied condition factor.

  $condition$ is a logical value of either true (1) or false (0). Because of this limitation, depth-first search algorithms are less practical and efficient. Breadth-first search algorithms are often more attractive.

- Nesting one gfor-loop inside another gfor-loop is not allowed. However, one gfor-loop can be nested among one or more nested regular for-loops.

- Memory allocation should be considered carefully. Since each operation in a gfor-loop is executed in parallel for all iterator values, sufficient GPU mem-

ory is required to support all iterations at the same time; otherwise, "out of memory" errors will occur.

- Subscripted data can not be cast back directly to CPU. On GPU, the parallel path run simultaneously with the same subscript, when these variables are pulled back to CPU, an extra dimension must be added to the destination matrix to avoid the subscript conflict. For example, if we need to pull a 4×4 parallel matrix product with 1024 parallel paths back to the CPU, a size $4 \times 4 \times 1024$ matrix should be prepared after the end of gfor-loop.
- Some of useful functions are unfortunately not supported inside a gfor-loop. Either new functions must be written, or the calculation will need to be re-structured.

The PC platform that we used to run experiments has a NVIDIA GeForce GTX 590 GPU with 1024 CUDA cores which are organized in 32 streaming multiprocessors of 32 cores each. The memory clock runs of 607 MHz. The standard memory configuration is 3073 MB and the memory bandwidth is 327.7 GB/sec.

We used a PC with an Intel (R) Core (TM) i7-2600k CPU running at a clock frequency of 3.40 GHz processor with 16.0 GB RAM. This CPU actually contains four independent cores that can each execute two parallel threads. In addition, these CPUs have a certain number of SIMD instructions for relatively simple vector arithmetic.

3.3 Review of Past Parallel Implementations of MIMO Detectors

Parallelism is often an efficient method to accelerate programs. In this section, we summarize past research on parallel implementations of MIMO detection algorithms.

Parallel Architecture of List Sphere Decoders (2007) [33]

The List Sphere Decoder (LSD) is a sphere decoder algorithm that searches a list

L which contains the most likely candidates with the smallest Euclidean distances (EDs). The parallel architecture of LSD can be divided into several parts: the first step is to compute the PED. This is done by a number of TSUs (Tree Search Units) in parallel. At the end of this operation, the results are written into cache memory. Then in the second step, a dispatcher unit finds the smallest PED, that is used to compare with the current radius when the leaf node is reached. If the new PED is smaller, the list is updated and a new sub-tree is assigned to the TSU. In [33], although the hardware was designed for a custom VLSI implementation, it still provides an example of parallel programming.

A Fixed-Complexity Sphere Decoder for MIMO Systems on Graphics Processing Units (2010) [34]

The fixed-complexity sphere decoder (FSD) is the main algorithm in this paper. To accomplish the parallelism, the author maps the FSD to the CUDA codes. In the simulation, both random input vectors are generated and the QR decomposition of channel matrix is executed directly in MATLAB. Then the CUDA C codes of the FSD are called in MATLAB to implement the detector. Since there are several memory types on the GPU, each variable of the FSD should be allocated properly. First, the data is copied from host memory to device memory, then the FSD is executed on the GPU device. With the CUDA-C environment, one instance of FSD will be mapped onto one thread on the GPU. Then the the degree of parallelism is determined by the number of threads that are created during the processing. The results from all threads will be transferred from device memory to host memory after all the processing has been finished. Compared to the normal C implementation for FSD, the speed-up increases rapidly with the help of the GPU.

Analysis of Parallel Sorting Algorithms in K-best Sphere Decoder Architectures for MIMO Systems (2011) [35]

As was mentioned in Chapter 2, the K-Best Sphere Decoder presents a trade-off between performance and complexity. Researchers have explored ways to reduce the computational complexity with the least reduction in performance. Parallel sorting

algorithms (PSA), which are proposed in this paper to deal with this problem, provide sorting strategies for the K-Best algorithm. The key structure in the PSA is an array of combined "interconnected Compare-and-Exchange cells". The inputs of this array are the corresponding branch-metric costs for each path at each layer, and the outputs are the sorted values that can be used in K-Best to determine the first K best symbols. PSA proposes to exploit customized hardware design (e.g., FPGA) to accomplish the parallelism of sorting.

Parallel SFSD MIMO Detection With Soft-Hard Combination Enumeration (2011) [36]

The acronym SFSD refers to a soft extension of conventional fixed-complexity sphere detection (FSD). The parallel SFSD (PSFSD) algorithm first generates the multiple detecting nodes simultaneously by applying ML detection to get partial best nodes. The search then proceeds layer by layer through the tree structure. In this way, only one tree searching operation is required. Instead of exhaustively searching according to full ML estimation, detecting nodes are first created with respect to the best partial ML estimate at the corresponding level of the tree to make sure they have accurate values with high probability.

Fully Parallel GPU Implementation of a Fixed-Complexity Soft-Output MIMO Detector (2012) [37]

Fully Parallel FSD (FPFSD) applies bit-interleaved coded modulation (BICM). FPFSD maintains different lists of candidates and distances in different channel layers to record soft information. Each layer has a different channel matrix column ordering to make sure that the top layers of the trees are different from each other, so that all the possibilities are obtained in the candidate paths. In this way, after all the data position information is available, each of the orders can be considered in parallel. The choice of column ordering strategy is quite important in this method. The norms of the channel's columns are calculated and sorted in ascending order. This order is the one needed to sort the rest of the channel layers. In this paper, the author uses CUDA to write the program which is also an another way to exploit the

parallel programming.

Parallel Processing Algorithm for Schnorr-Euchner Sphere Decoder (2012) [38]

In this paper, the SE Sphere Decoder, which is a well-known depth-first search algorithm, is modified through a complex-to-real conversion step to simplify parallel processing. This complex-to-real conversion is similar to the equation Eq. (2.7) which was described in Chapter 2, but there are some important differences. Instead of simply adding the real and imaginary parts, the MIMO system equation is changed as follows:

$$\begin{bmatrix} \Re(\tilde{y}_1) \\ \Im(\tilde{y}_1) \\ \vdots \end{bmatrix} = \begin{bmatrix} \Re(\tilde{h}_{11}) & -\Im(\tilde{h}_{11}) & \cdots \\ \Im(\tilde{h}_{11}) & \Re(\tilde{h}_{11}) & \cdots \\ \vdots & \ddots & \end{bmatrix} \begin{bmatrix} \Re(\tilde{s}_1) \\ \Im(\tilde{s}_1) \\ \vdots \end{bmatrix} + \begin{bmatrix} \Re(\tilde{n}_1) \\ \Im(\tilde{n}_1) \\ \vdots \end{bmatrix} \tag{3.2}$$

In this way, the $2k$-th layer and $(2k-1)$-th layer for $k = 1, 2, \cdots, M_t$, which represent the real and imaginary components of detected symbol, respectively, are independent of each other so that the search path can be arranged simultaneously between these two nodes which are defined as a "node pair" in this paper. The Parallel Sphere Decoder (PSD) algorithm moves from node pair to node pair to reduce the computing time.

Depth-first search algorithms can take an irregular path when trying to find the optimal path from the root node to the leaves, so they are often considered to be hard to synchronize in a parallel implementation. In this paper, the author provides a good idea ("node pairs" that can be traversed in parallel) that bring the parallelism into the Depth-first searching.

Conclusion of Previous Research on Parallel Programming Since MIMO detection algorithms are very useful in our daily life, and also parallel implementations can help to accelerate the algorithm efficiently, this topic has attracted interest among many researchers. From the papers we summarized previously, the approaches can be separated into two groups, software implementations and hardware implementations.

Software implementations mostly use different programming language on the existing parallel enabled hardware. The parallel enabled hardware can be workstations with multicore CPUs or GPUs, the researchers do not need to design the hardware but have to be familiar with the development programming languages such as C/C++, CUDA C for GPU, etc. They came up with several different ideas to build the data structures to fit the parallelism models for different detection algorithms. But because the hardware environment is fixed, there are also varies limitations during the software programming.

Hardware implementation requires more knowledge about the hardware design, but as a benefit, the structure of the detection algorithms can be more flexible to the hardware. The commonly used hardware environments are FPGA and the VLSI. Researchers can point each data structure or even a single detection algorithm package to a unit on the chip, allocate the different memory to different working space and trace and control the parallelism step by step.

∼

Chapter 4

Parallel Implementation of MIMO Detection Algorithms on the GPU

In Chapter 3, we reviewed the key aspects of GPU technology and parallel programming. General-purpose GPUs have already been applied in several different areas [39]. Our research aims to speed up the standard MIMO detection algorithms by exploiting the hardware parallelism of the GPU and the parallel computing environment provided by the Jacket extension of MATLAB. To ensure the efficiency of the parallel approach, most of data should be generated and processed in parallel on the GPU to avoid time-consuming transfers of data between the CPU and GPU. This means that we need to rewrite conventional MATLAB MIMO detector models using the Jacket library functions to ensure the parallel operation of the GPU based detection programs up to the parallelism limits of the underlying hardware.

In a MIMO system, we aim to process more data streams in a shorter time to gain higher efficiency. If these data streams can be efficiently mapped in a directly scalable way onto a parallel structure and processed at the same time, then acceleration can be achieved by increasing the number of parallel paths.

4.1 Matrix Multiplication in Parallel

To achieve the greatest acceleration performance in light of Amdahls Law, we should in general parallelize as many parts of the algorithm as possible. Matrix multiplication is a critical operation in our algorithms, for example, in the V-BLAST algorithm. After the interference cancellation step at each layer, the matrix inverse

Code Listing 4.1: Source code for the Matrix Multiplication with a conventional MATLAB for-loop and the Jacket gfor-loop

```
C1 = gzeros(N,N,Parallelism);   % N is the size of matrix
C2 = gzeros(N,N,Parallelism);   % Parallelism is the degree of
   parallelism
for outloop = 1:100
    A = grand(N,N,Parallelism);
    B = grand(N,N,Parallelism);
    Bt = B';   % Transpose needed for the dot product in for-loop
    % for-loop applied
    for ii = 1:N
        for jj = 1:N
            C1(ii,jj,:) = dot(conj(A(ii,1:N,:)),Bt(jj,1:N,:));
        end
    end
    % gfor-loop applied
    gfor pp = 1:Parallelism
        C2(:,:,pp) = A(:,:,pp)*B(:,:,pp);
    gend
end
```

(which will be discussed in detail in Section 4.4.1) is always required, and the matrix multiplication costs most of the time during the calculation and it will be the bottleneck of the acceleration of this algorithm. So we decided to conduct experiments to determine the best way to implement this critical operation in parallel. In parallel matrix processing our data structures are often three dimensional, where the first two dimensions correspond to the number of rows and columns and the third dimension corresponds to the degree of parallelism.

4.1.1 Experiment 1 for the for and gfor Looping Structures

Two alternative methods are compared in this experiment. The MATLAB source code used in the experiment is shown as Code Listing 4.1. The first method uses two nested for-loops to do the dot product on each row and column vectors of the two input matrices. The second method uses a single gfor-loop from the Jacket library as the inner loop. The usage of the gfor-loop is almost the same as the for-loop in MATLAB; the only difference is that the iterator in a gfor specifies the degree of parallelism across GPU cores. The operations in a gfor loop can be viewed as executing in parallel on different streams of data in SIMD fashion.

The experimental results are shown in Table 4.1. where $N \times N$ is the size of the

Table 4.1: Matrix multiplication mimes (in seconds) for different looping (for and gfor) structures

Degree of Parallelism	for-loop		gfor-loop	
	$N = 4$	$N = 8$	$N = 4$	$N = 8$
128	0.58	2.29	0.07	0.06
256	0.58	2.29	0.11	0.10
512	0.58	2.32	0.19	0.19
1024	0.58	2.36	0.34	0.37
2048	0.58	2.36	0.68	0.75
10240	0.61	2.68	3.54	3.57

real-valued matrices. The table gives the average running times (in seconds) of real-valued matrix multiplication based using the for-loop and gfor-loop constructs. For a reliable measurement, we repeated the test 100 times using the outer for-loop, and so these running time are amplified 100 times greater than a single matrix multiplication. It can be seen that the running time is not greatly influenced by the increases in the matrix size in the gfor-loop implementation while it causes a big impact in the for-loop implementation. In other words, when the degree of parallelism increases, the running time of the for-loop method keeps almost steady, while for the gfor-loop method the running time increases directly at the same rate as the degree of parallelism. However, it is clear that even though the gfor-loop's running time increases, it is still faster than the for-loop, until the degree of parallelism reaches to 1024, which is the number of GPU cores. Moreover, for the gfor-loop method, the size of the matrix doesn't affect the running time of multiplication, while it quadruples for the for-loop.

4.1.2 Experiment 2 for the Serial and Parallel gfor Looping Structures

Having compared the different loop models, we also wanted to determine how much improvement we could achieve from parallelism compared with serial multiplica-

Code Listing 4.2: Source Code for the Matrix Multiplication Experiment with Serial and Parallel Versions

```
% Serial version on the CPU
for ii = 1:Parallelism*100
    A = rand(N,N);
    B = rand(N,N);
    C(:,:,ii) = A*B;
end

% Parallel version on the GPU
for outloop = 1:100
    AA = grand(N,N,Parallelism);
    BB = grand(N,N,Parallelism);
    CC = gzeros(N,N,Parallelism);
    gfor pp = 1:Parallelism
        CC(:,:,pp) = AA(:,:,pp)*BB(:,:,pp);
    gend
end
```

tions for different sizes of matrices. In the parallel version, we apply the gfor-loop structure for the multiplication, while in the serial version, a for-loop with an iterator equals serially the degree of parallelism is used so that the multiplication can be executed in serial.

The source code for this second experiment is shown in Code Listing 4.2, where N stands for the size of a matrix and $Parallelism$ is the degree of parallelism. In order to get an equivalent result, the number of iterations is set to be $Parallelism \times 100$ in the serial version.

Table 4.2 shows the results from this test. As in Table 4.1, the performance is measured by the running time (including 100 outer loop repetitions) of each version. The "Speed-Up" values are calculated as:

$$\text{Speed-Up} = \frac{\text{Time for the serial version}}{\text{Time for the parallel version}} \tag{4.1}$$

Results could not be obtained when the size of matrix grows to 128 and the degree of parallelism equals to 10240. Jacket is unable to allocate sufficient memory from GPU to do the multiplications under these conditions. The serial multiplication time grows rapidly when the size of the matrix increases but the running time for the parallel version remains relatively constant. It is only impacted by the in-

Table 4.2: Matrix multiplication times (in seconds) for serial and different degrees of parallel versions

Size of Matrix N	Degree of Parallelism								
	512			1024			10240		
	Serial	Parallel	Speed-up	Serial	Parallel	Speed-up	Serial	Parallel	Speed-up
4	0.18	0.16	1.13	0.38	0.34	1.12	3.65	3.35	1.09
8	0.25	0.17	1.47	0.49	0.33	1.48	4.78	3.35	1.43
16	0.33	0.17	1.94	0.65	0.33	1.97	6.41	3.35	1.91
32	0.79	0.17	4.65	1.55	0.33	3.48	15.50	4.33	3.58
64	2.85	0.18	15.83	5.67	0.37	15.32	56.89	13.70	4.15
128	11.71	0.68	17.22	23.46	1.49	15.75	231.93	-	-

creasing number of parallelism. The reason for this is that the GPU has its own coordinate system and structure (as described in Chapter 3), to ensure the simultaneous operations on all the elements of a matrix during the computation [40].

It is clear in Table 4.1 that when the matrix size of each parallel path is small, we can not take much advantage of the larger parallelism in the GPU. The larger the matrix size becomes, the much more speed-up we can get from the parallelism. This is because the matrix multiplication in MATLAB itself has already taken advantage of the multithreading technology on the CPU and the CPU's clock speed is much faster than GPU's, and the overhead time to use the GPU should also be counted in the calculation. Then when the matrix size is small, the acceleration can be rarely seen from this test.

4.1.3 Experiment 3 for Merged Matrix Multiplication with Parallel gfor-loop

The results from the previous Experiment 2 showed that when the matrix size in each of parallel path grows bigger, we can achieve more acceleration from the parallelism. So we decided to try merging multiple small matrices into one large matrix to see how much acceleration and advantage could be obtained.

The strategy of this experiment is to merge small matrices into the diagonal of a large matrix. Taking two groups of four small 4×4 matrices **A**, **B**, **C**, **D**, **E**, **F**, **G** and **H** as an example, the multiplication equation is shown in Eq. (4.2).

$$\begin{bmatrix} \mathbf{A}\,0\,0\,0 \\ 0\,\mathbf{B}\,0\,0 \\ 0\,0\,\mathbf{C}\,0 \\ 0\,0\,0\,\mathbf{D} \end{bmatrix} \times \begin{bmatrix} \mathbf{E}\,0\,0\,0 \\ 0\,\mathbf{F}\,0\,0 \\ 0\,0\,\mathbf{G}\,0 \\ 0\,0\,0\,\mathbf{H} \end{bmatrix} = \begin{bmatrix} \mathbf{AE} & 0 & 0 & 0 \\ 0 & \mathbf{BF} & 0 & 0 \\ 0 & 0 & \mathbf{CG} & 0 \\ 0 & 0 & 0 & \mathbf{DH} \end{bmatrix} \tag{4.2}$$

where **AE**, **BF**, **CG** and **EH** stands for the sub-matrix products $\mathbf{A} \times \mathbf{E}$, $\mathbf{B} \times \mathbf{F}$, $\mathbf{C} \times \mathbf{G}$ and $\mathbf{E} \times \mathbf{H}$, respectively.

It can be seen from Eq. (4.2) that these four ssub-matrix multiplications are executed at the same time in a large matrix to save running time. In this equation, we set the size of matrix to be 4 which could also be changed. In this experiment, we deal with the square matrix with the size of N. However, the matrix does not have to be square, only if two small matrices at each multiplication side can be

Code Listing 4.3: Source Code for the Merged Matrix Multiplication Experiment with the Parallel gfor-loop Structure

```
Parallelism = 1024 % Degree of parallelism
N = 4;    % Matrix size
F = 1;    % Number of component sub-matrices
interval = N-1;   % Number of rows/columns between each small
   matrix
for loop = 1:100
    LeftMatrix  = gzeros(N*F,N*F,Parallelism);
    RightMatrix = gzeros(N*F,N*F,Parallelism);
    ProdMatrix = gzeros(N*F,N*F,Parallelism);
    for f = 1:F
       LeftMatrix((f*N-interval):f*N,(f*N-interval):f*N,:) = grand
          (N,N,Parallelism);
       RightMatrix((f*N-interval):f*N,(f*N-interval):f*N,:) =
          grand(N,N,Parallelism);
    end
    gfor pp = 1:Parallelism
        ProdMatrix(:,:,pp) = LeftMatrix(:,:,pp) * RightMatrix(:,:,
           pp);
    gend
    for f = 1:F
        AE = ProdMatrix((f*N-interval):f*N,(f*N-interval):f*N,:);
           % result for each small matrix
    end
end
```

matched and put into the diagonal of two large matrices. This is strategy is also applied in the parallel implementation of MIMO detection algorithms later.

The source code of this experiment is in Code Listing 4.3

In Code Listing 4.3, $Parallelism$ is the degree of parallelism, which is set to be 1024. N is the size of small square matrix. F is the number of small matrices that have been combined into one matrix. F can also be seen as a speed-up factor for the $N \times N$ matrix within the $NF \times NF$ matrix. The same as in Experiments 1 and 2, we also set an outer loop to repeat all the operations.

The results of this experiment is shown in Table 4.3.

In Table 4.3, the data in columns “gfor” show the running times of the gfor-loop (including 100 outer loops repetitions), which only contain the matrix multiplication inside. The data in columns “Speed-up” compare the time for different value of F with “$F = 1$” for each N. The data in $F = 73$ with $N = 4$, $F = 36$ with $N = 8$, $F = 18$ with $N = 16$, $F = 4$ with $N = 64$, $F = 2$ with $N = 128$ and

Table 4.3: Matrix multiplication times (in seconds) for the merged matrix with parallel gfor-loop structure

F	Matrix Size N											
	4		8		16		64		128		256	
	gfor	Speed-up	gfor	Speed-up	gfor	Speed-up	gfor	Speed-up	gfor	Speed-up	gfor	Speed-up
1	0.34	1.00	0.34	1.00	0.34	1.00	0.40	1.00	1.64	1.00	15.97	1.00
2	0.33	2.06	0.35	1.94	0.34	2.00	0.34	2.35	15.56	0.21	Out of Memory	
3	0.34	3.00	0.35	2.91	0.33	3.09	0.39	3.08	Out of Memory			
4	0.33	4.12	0.34	4.00	0.33	4.12	15.28	0.10				
5	0.34	5.00	0.34	5.00	0.34	5.00	Out of Memory					
10	0.34	10.00	0.34	10.00	0.44	7.73						
17	0.35	16.51	0.54	10.73	19.63	0.29						
18	0.36	17.00	0.56	10.93	44.12	0.14						
20	0.37	18.38	0.56	12.14	Out of Memory							
30	0.53	19.25	1.22	8.36								
35	0.81	14.69	21.59	0.55								
36	0.81	15.11	43.62	0.28								
50	1.40	12.14	Out of Memory									
70	21.76	1.09										
73	46.89	0.53										
74	Out of Memory											

$F = 1$ with $N = 256$ are the maximum limits of each matrix size. When the the size grows over that, the system runs out of memory. These data are not fixed, they depend on the total amount of the available memory on the GPU.

It can be seen from Table 4.3 that the speed-up of the merged matrix multiplication is significant. For each matrix size N, the running times for multiplication in gfor-loop stay almost the same when F increases, until the available device memory reaches to the end. The results from this experiment provide a useful reference for the parallelism model that can be chosen in next sections.

4.2 Models of Parallelism

Since the GPU in our host PC has 1024 cores, the degree of parallelism was set to be 1024 in our experiments to keep all cores busy during the processing. According to the test results in Table 4.1 in Experiment 1 in the last section, when the size of the matrix increases, the gfor-loop method will take greater advantage of the parallelism than the nested for-loop method, so we apply the gfor-loop structure in our parallelism model.

Algorithm 1 gives the general parallel structure that we investigated in this research:

Algorithm 1 Parallelism Model

1: Set the Total number of the outer loops $TotalLoop$
2: **for all** $outloop = 1 : TotalLoop$ **do**
3: $\quad M_t \times 1 \times NumParallel$ random symbol vectors are generated
4: $\quad M_r \times M_t \times NumParallel$ random channel matrices are generated
$\quad$ (10 symbol vectors are processed for each new channel)
5: $\quad M_r \times 1 \times NumParallel$ Gaussian noise samples are generated
6: $\quad$ **gfor** $pp = 1 : NumParallel$
7: $\quad\quad$ Detection algorithm is applied on $NumParallel$ symbols in parallel
8: $\quad\quad$ $NumParallel$ symbols are detected in parallel
9: $\quad$ **gend**
10: **end for**

In Algorithm 1, $TotalLoop$ denotes the repetitions that a detection algorithm

needs to do. $NumParallel$ stands for the number of parallel GPU threads that can be executed simultaneously. M_t and M_r stand for the number of antennas at the transmitter and the receiver, respectively, following the same convention used in Chapter 2. In this way we detect $TotalLoop \times NumParallel$ symbol vectors by the end of program execution and hopefully reduce the running time by efficiently exploiting the hardware parallelism.

Since we obtain fairly good results from Experiment 3 in Section 4.1.3, we can also have another parallelism model by applying the strategy of merged matrix multiplication to detection algorithms to see how much advantages we can take.

Algorithm 2 Parallelism Model Using Merged Matrix

1: Set the merge factor F (refer to the results in experiment 3 in Section 4.1.3)
2: Set the Total number of the outer loops $TotalLoop$
3: **for all** $outloop = 1 : TotalLoop/F$ **do**
4: $M_t * F \times 1 * F \times NumParallel$ random symbol vectors are generated
5: $M_r * F \times M_t * F \times NumParallel$ random channel matrices are generated (10 symbol vectors are processed for each new channel)
6: $M_r * F \times 1 * F \times NumParallel$ Gaussian noise samples are generated
7: **gfor** $pp = 1 : NumParallel$
8: Detection algorithm is applied on $NumParallel * F$ symbols in parallel (The degree of parallelism is still $NumParallel$)
9: $NumParallel * F$ symbols are detected in parallel
10: **gend**
11: **end for**

It can be seen from Algorithm 2 that the structure of the model is almost the same as Algorithm 1, the difference is that the data matrix is amplified by the factor F, which was introduced in Experiment 3 in Section 4.1.3, to enable F times matrices/vectors to be operated on at the same time. Line 3 shows the reduction of the outer loops if we have a fixed total amount ($TotalLoops \times NumParallel$) of symbols vectors when the factor F is applied.

The performance of the serial and parallel versions is compared in Table 4.5.

4.3 Channel Generation on the GPU

Since our simulation model as described in Chapter 2, generates the channel and noise using the MATLAB's built-in function "randn", the first step of efficient parallelization is to generate all these signals on the GPU. It is important for efficiency to avoid moving data between the CPU and GPU as well as between GPU cores. As much as possible, data should be generated and processed in parallel within the GPU cores. In the Jacket library, there are many useful functions that can achieve this task. The Jacket function "grandn" is used to generate normally distributed pseudo-random numbers on the GPU. Both the channel coefficients and the additive white Gaussian noise samples are generated using "grandn". Symbol generation must be done differently because MATLAB's built-in functions "randi" and "qammod" are not supported with parallel versions on the GPU. These two functions are used to generate integer values from the uniform distribution and produce a random stream of QAM symbols. The random bit stream is encoded using a Gray Code in the real and imaginary dimensions, following standard practice, to minimize the number of bit errors produced by symbol detection errors during symbol detection. For most symbol detection errors, only one bit error will be produced; only rarely will two or more bit errors occur because of one symbol error.

The distributions of these three generated values are shown in Fig. 4.1. Note that two independent 4-PAM symbols are required for each complex 16-QAM symbol.

As can be seen in Fig. 4.1, the distributions of both the noise samples and the channel coefficients accurately follow the Gaussian distribution.

4.4 Parallel Implementation of MIMO Detection Algorithms

In Chapter 2, we listed three linear detection algorithms and three sphere detection algorithms. In order to efficiently parallelize these algorithms, functions and data streams must be implemented efficiently on the GPU. So we had to carefully modify the existing MATLAB implementations based on the documented strengths of

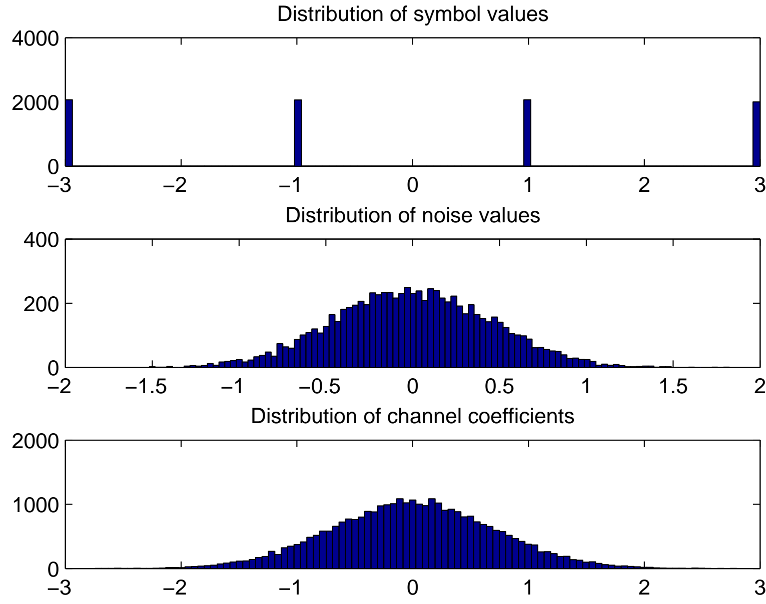

Figure 4.1: Distribution of 4-PAM symbols, additive noise and MIMO channel coefficients

Jacket [32].

4.4.1 Modification of Channel Inversion

For the first three linear detection algorithms, the most complex calculation is to compute the Moore-Penrose pseudo inverse of the channel ($\tilde{\mathbf{H}}^{\dagger}$), which was described in Chapter 2 in Eqs. (2.12) and (2.13). In conventional serial MATLAB, the inverse of a matrix can be implemented using the built-in function "inv". However, we found the running time to be quite large using this function: it costs almost half of the time of a linear MIMO detection program. Note that in the V-BLAST algorithm, we have to apply interference cancellation after each layer's slicing and quantization steps, and so the resulting channel matrix inverse calculations become the bottleneck of the detector simulation. So we decided to design an improved function to accomplish the matrix inverse.

Our new function "NewInverse" performs LU decomposition and then solves the resulting linear equations. Assume that there is an $N \times N$ matrix $\mathbf{A}$ and a linear

equation $\mathbf{AX} = \mathbf{B}$. If $\mathbf{B}$ is set to be an $N \times N$ identity matrix, then $\mathbf{X}$ must be the inverse matrix of $\mathbf{A}$. The matrix inverse decomposition proceeds as follows:

1. Apply LU decomposition on $\mathbf{A}$ and get a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$ such then $\mathbf{A} = \mathbf{LU}$. The linear equation $\mathbf{AX} = \mathbf{B}$ can then be rewritten as $\mathbf{LUX} = \mathbf{B}$.

2. Solve the equation $\mathbf{LY} = \mathbf{B}$, where $\mathbf{Y} = \mathbf{UX}$.

3. Solve the equation $\mathbf{UX} = \mathbf{Y}$ for $\mathbf{X}$.

In steps 2 and 3, the equations can be easily solved by forward and backward substitution N times without using Gaussian elimination because of the triangular forms of matrices $\mathbf{L}$ and $\mathbf{U}$.

Table 4.4: Comparison of matrix inverse runing times (in seconds) using built-in function "inv" and new function "NewInverse"

Size N of Matrix	Built-in inv(**A**)	NewInverse(**A**)	Matrix Division	
			$\mathbf{A} \backslash \mathbf{B}$	NewInverse($\mathbf{A} * \mathbf{B}$)
4	0.297	0.078	0.277	0.078
8	0.356	0.396	0.291	0.419
16	0.503	1.793	0.351	1.811
32	0.741	7.737	0.513	7.599
64	1.3071	32.119	0.775	29.206

The brief comparison between this new inverse version ("NewInverse") and MATLAB built-in function "inv" is given in Table 4.4. $\mathbf{A}$ and $\mathbf{B}$ are two random square matrices of size N. Since the purpose of matrix inverse in this thesis is to do the matrix division for the channel (as Eq. (2.12) and Eq. (2.13)), the matrix division is also included in this comparison. In MATLAB, instead of using $inv(\mathbf{A}) * \mathbf{B}$ for $\mathbf{B}/\mathbf{A}$, $\mathbf{A} \backslash \mathbf{B}$ is more efficient and faster according to the documentation of MATLAB. So the matrix division is compared between $\mathbf{A} \backslash \mathbf{B}$ and NewInverse($\mathbf{A} * \mathbf{B}$).

The results in Table 4.4 shows that when the matrix size is small, our new inversion function is much faster than the MATLAB built-in function “inv”, but when the size increases, we should decide which method to be applied depending on different situations. In this thesis, when a 4×4 real-valued matrix inverse need to be considered, the speed up provides enough improvement during the processing.

4.4.2 Parallel Versions of the Linear MIMO Detection Algorithms

The major strategy in parallelization is to make sure that all the data structures are initialized on the GPU before the detection process begins and are then updated in parallel on the GPU. In this way, we can minimize time-consuming data transfers between the CPU and GPU. As described in the previous section, the transmitted signals, channel matrices and noise signals have already been loaded on the GPU, and the performance of the channel inverse calculation has also been improved. We can now directly implement parallel versions of the detection algorithms. Since these algorithms are implemented only by slicing, quantization and interference cancellation (e.g., the V-BLAST algorithm) and all these operations can be fully supported on the GPU, it is relatively straightforward to convert them into fully parallel versions using Jacket functions.

The first part of our research is to test the performance of our algorithms on the GPU. The running times of serial and parallel versions are compared in Table 4.5.

The assumed system environment is as follows:

- 4×4 MIMO System
- Modulation type: 16-QAM
- Number of symbols processed: 1000×1024 symbol vectors
- SNR = 20 dB

In Table 4.5, the data in the columns of the “Serial” and “GPU Parallel gfor” are the running time (in seconds) of each algorithm in both versions, respectively. The running times include all 1000×1024 symbol vectors. The speed-up is calculated as in Eq. (4.1).

Table 4.5: Running times (in seconds)comparison of MIMO detection algorithms with the serial and different parallel versions

	Serial (1024000)	GPU Parallel gfor (1000×1024)	Speed-Up	GPU Parallel gfor, $F = 18$ ($18 \times 55 \times 1024$)	Speed-Up	GPU Parallel gfor (10×102400)	Speed-Up
Channel Generation	29.574	0.107	276.393	0.311	95.093	0.017	1770.900
Data Generation	39.879	0.670	59.521	1.086	36.721	0.172	231.855
ZF Detection	278.603	15.282	18.231	6.552	42.522	8.030	34.695
MMSE Detection	287.134	15.296	18.772	6.572	43.691	8.044	35.695
V-BLAST Detection	626.020	191.333	3.272	133.290	4.697	64.530	9.701
K-Best Detection	703.008	620.127	1.134	-	-	490.303	1.434
Parallel V-BLAST Detection	3075.281	513.076	5.994	234.749	13.100	290.964	10.569

The notation 1000×1024 means that the program loops 1000 times while 1024 parallel signal paths are processed concurrently in each loop iteration (according to the Algorithm 1 described above). Thus the notation 10×102400 corresponds to a program that loops 10 times while 102400 parallel signal detector paths are processed each time. "Data Generation" stands for the generation of all the transmitted symbols, channel matrices and noise samples. Most of the data are complex-valued, however the K-Best algorithm deals with real-valued data.

It can be seen from Table 4.5 that the speed-up factor from the normal parallel gfor loop (1000×1024) is not as good as we expected. There must be some overhead during the processing, especially since the data matrix of each parallel path is quite small so that we can not achieve much benefits from the parallelism. When the degree of parallelism increases to 102400, the improvement becomes better. The results from the K-Best algorithm are quite the same, there is little speed-up going from the serial to normal parallel version. It's because in the K-Best algorithm, most of the operations are applied node by node at each level, and there are few matrix multiplications. The CPU's speed of processing for one single data is already fast enough so that the GPU calculation can not take much advantage from that.

The speed up results in columns 5 and 6 with a factor F are different than for the other parallel versions. In order to understand the limits to acceleration, the top 3 most time consuming parts of each algorithms are listed in Table 4.5. We ran the profiler in MATLAB and listed in the Table 4.6.

According to Table 4.6, it can be seen that for all four linear detection algorithms, matrix multiplication and matrix inverse are the most time consuming parts during the processing. To solve the first problem, merged matrix multiplication can be applied to provide a significant improvement, which can be seen in Table 4.5 column 5 and 6 with $F = 18$.

The speed-up factors of the ZF and MMSE detection algorithms are about 35x, which are also the number of the matrix multiplication time consuming percentage. For the V-BLAST detection algorithm, the bottleneck is the channel matrix inverse. Note that in this V-BLAST version we have already modified the matrix inverse in Section 4.1.1 and Table 4.4 shows that larger matrix costs more time to inverse,

Table 4.6: The most time consuming operations for the MIMO detection algorithms

Algorithm	Instructions	Percentage
ZF Detection	$\mathbf{tempVec} = \mathbf{G}_{ZF}\mathbf{y}$ (Nulling)	40.1%
	$\mathbf{AfterInverseMat} = \text{MatrixInverse}(\mathbf{H})$	19.7%
	$\mathbf{y} = \mathbf{Hs} + \mathbf{n}$	17.5%
MMSE Detection	$\mathbf{tempVec} = \mathbf{G}_{MMSE}\mathbf{y}$ (Nulling)	40.3%
	$\mathbf{AfterInverseMat} = \text{MatrixInverse}(\mathbf{H})$	19.6%
	$\mathbf{y} = \mathbf{Hs} + \mathbf{n}$	17.7%
V-BLAST Detection	$\mathbf{AfterInverseMat} = \text{MatrixInverse}(\mathbf{H})$	70.0%
	$\mathbf{InverseMat} = \mathbf{H}^H\mathbf{H} + (1/snr) * \mathbf{I}_n)$	8.2%
	$\mathbf{G}_{VBLAST} = \mathbf{AfterInverseMat} \times \mathbf{H}^H$	8.0%
K-Best Detection	Calculate the partial Euclidean distance (PED)	72.1%
	Locate the first K nodes of each level	16.8%
	Load the detected symbol nodes for previous level	3.6%
Parallel V-BLAST Detection	$\mathbf{tempVec} = \mathbf{G}_{VBLAST}\mathbf{y}$ (Nulling)	25.9%
	$\mathbf{AfterInverseMat} = \text{MatrixInverse}(\mathbf{H})$	19.3%
	$\mathbf{tempY} = \mathbf{H} \times \mathbf{symbolTest}$	8.4%

so we can not take much advantage of merging small matrices together even if the built-in function “inv textquotedblright is applied. Then the speed-up factor is affected by the amount of matrix multiplications in the V-BLAST algorithm.

In the K-Best detection algorithm, we can take more advantages of adding the factor F to the program. Because in each step, the operations work on one node, it rarely requires the matrix multiplication. If we put several $2M_r \times 2M_t$ (real-valued) matrices into a large matrix, we still have to evaluate the nodes (detect the symbols) level by level, and even the overhead of using “for f = 1:F ” loop will cost other time. Then there’s no data for K-Best using F-factor parallelism in Table 4.5.

The speed-up of the parallel V-BLAST detection algorithm using the merged matrix strategy is better than for the normal parallel version (1000×1024) from Table 4.6. Since the weakest channel layer is fully enumerated with all the constellation points, the operations of the V-BLAST algorithm are repeated 16 times if 16-QAM is applied during the processing. By packing F matrices together, the total amount of 16-times-repeated matrix multiplications is reduced efficiently. Note that, from the matrix inverse result in Table 4.4, we also use MATLAB built-in function “\ ” to do the matrix division in this Parallel V-BLAST version.

4.4.3 The Parallel V-BLAST Algorithm

In the original V-BLAST algorithm, the first detected layer is chosen to be the layer with the minimum norm and hence the lowest expected post-detection SER. After symbol detection, we subtract the predicted contribution of that symbol on the signal vector (interference cancellation) to minimize the SER on the remaining symbols to be detected. Errors in the detection of the first layer increases the interference in the detection of the following layers. The parallel V-BLAST algorithm in [18] tries to avoid this effect by fully enumerating the weakest layer to minimize detection errors in the strongest layer. In this way, in the first “detected ” layer all 16 possible symbol values (16-QAM) of the weakest layer will be considered, and then the original V-BLAST detector will be applied on the remaining layers. At the end of this algorithm, 16 candidate detected symbol vectors are compared and only the one with the minimum Euclidean distance between the predicted noise-free signal

Hs and the received symbol vector **y** is picked as the detected symbol vector.

Algorithm 3 The Parallel V-BLAST Algorithm

Inputs:

The numbers of transmitter and receiver antennas M_t, M_r, respectively;

16-QAM Constellation set **ConsMat** of size ConsSize;

The channel matrix $\tilde{\mathbf{H}}$, the symbol vector $\tilde{\mathbf{s}}$ and the received signal vector $\tilde{\mathbf{y}}$;

The Moore-Penrose pseudo-inverse matrix $\tilde{\mathbf{G}}$;

Output:

The number of symbol errors from the detector;

1: $\text{layer}_{\text{weakest}} = \max(\text{norm}(\tilde{\mathbf{G}}))$
2: **for all** $i = 1 : \text{ConsSize}$ **do**
3:     $\text{DetSym}(\text{layer}_{\text{weakest}}) = \text{ConsMat}(i)$
4:     Cancel the interference from the $\text{layer}_{\text{weakest}}$-th layer
5:     Apply the V-BLAST algorithm to detect the remaining layers
6: **end for**
7: Combine 16 candidate symbol vectors as a matrix $\tilde{\mathbf{DetSym}}$
8: $\text{BestSetIndex} = \min(\|\tilde{\mathbf{y}} - \tilde{\mathbf{H}} \times \tilde{\mathbf{DetSym}}\|^2)$
9: Compare $\tilde{\mathbf{DetSym}}(\text{BestSetIndex})$ to $\tilde{\mathbf{s}}$, calculate the number of symbol errors

---

In Algorithm 3, the channel, symbol and received symbol vectors use the same complex-valued convention (Eq. (2.5)) used in Chapter 2. $\tilde{\mathbf{G}}$ equals $\tilde{\mathbf{G}}_{MMSE}$ as shown in Eq. (2.13) in Chapter 2. **ConsMat** denotes the 16-QAM constellation set $\{-3-3i, -3-1i, \cdots, 3+3i\}$ shown in Fig. 2.2 in Chapter 2.

Ideally, all 16 candidate symbol vectors should be processed in parallel to get the maximum speed-up compared to the serial V-BLAST algorithm. However, as we mentioned in Chapter 3, the Jacket library uses the gfor loop to specify one explicit dimension of parallelism and Jacket does not allow nested gfor loops. We have already set 1024 parallel paths to run the program at the beginning, so it is impossible for us to set another 16 parallel paths within 1024 paths. However it could be possible to have 64×16 parallel execution paths. Our parallel V-BLAST implementation processed the 16 symbols in the first/weakest layer one by one to get the 16 candidate detected symbol vectors. The performance of this parallel V-BLAST can be seen in Fig. 4.2.

It can be seen from Fig. 4.2 that the parallel V-BLAST algorithm's performance is near-optimal. By enumerating all possible values of the weakest symbol we remove a significant source of interference noise on all other symbols. Significantly, detection errors on the strongest symbol are reduced, and this reduces error propagation to detection errors affecting the detection of the other symbols.

4.4.4 Parallel V-BLAST with Real and Imaginary Components

This algorithm modified the original parallel V-BLAST algorithm [41]. The main modification is that the real and imaginary components are treated separately and all calculations are real-valued. For 16-QAM, 16 possible weakest-layer symbols are enumerated in the original complex-valued algorithm. But by treating the real and imaginary components separately, only 4 possible component-values need to be considered for each component and so the total number of candidates is reduced from $M_c = 16$ to $2\sqrt{M_c} = 8$. Therefore, the computational complexity can be partially reduced compared to the complex-valued parallel V-BLAST, the number of candidate symbols calculations is reduced while the matrices are bigger (from 4×4 to 8×8) which increases the cost of matrix inversions and multiplications. This advantage could be significant for large MIMO systems with big number of antennas. The algorithm shown in Algorithm 4.

In Algorithm 4, the channel, symbol and received symbol vectors use the real-valued convention as Eq. (2.7) in Chapter 2. $\tilde{\mathbf{G}}$ still uses the calculation of $\mathbf{G}_{MMSE}$ as Eq. (2.13) in Chapter 2. **RealConsMat** is simplified to $\Omega = \{-3, -1, 1, 3\}$ which was also introduced in Chapter 2, and its size is reduced to RealConsSize $= M_c = 4$. The detection loop iterates 8 times, four times for the real and imaginary values of the weakest layer. Line 2 indicates the related imaginary component's layer when the weakest real component's layer is determined in line 1.

As with real-valued parallel V-BLAST, we did not make these 8 candidates into parallel threads. It could also be possible to have a 128×8 parallel execution model. Fig. 4.2 shows the performance of three V-BLAST algorithms compared to the ML detection.

As can be seen from Fig. 4.2, the real-valued parallel V-BLAST detection al-

Algorithm 4 Parallel V-BLAST with Real and Imaginary Components

Inputs:

The numbers of transmitter and receiver's antennas M_t, M_r;

16-QAM Constellation set **RealConsMat** and its size RealConsSize;

The $2M_r \times 2M_t$ real-valued channel matrix **H**, the real-valued symbol vector **s** and the real-valued received signal vector **y**;

The real-valued Moore-Penrose pseudo inverse matrix **G**;

Output:

The number of symbol errors from the detector;

1: $\text{Reallayer}_{\text{weakest}} = \max(\text{norm}(\mathbf{G}))$
2: $\text{Imaglayer}_{\text{weakest}} = \text{Reallayer}_{\text{weakest}} + \text{RealConsSize}$
3: **for all** $i = 1 : \text{RealConsSize}$ **do**
4: $\quad$ $\text{RealDetSym}(\text{Reallayer}_{\text{weakest}}) = \text{RealConsMat}(i)$
5: $\quad$ Interference cancellation on $\text{Reallayer}_{\text{weakest}}$-th layer
6: $\quad$ Normal V-BLAST algorithm on the remaining layers
7: **end for**
8: Combine 4 candidate symbol vectors as a matrix **RealDetSym**
9: **for all** $j = 1 : \text{RealConsSize}$ **do**
10: $\quad$ $\text{ImagDetSym}(\text{Imaglayer}_{\text{weakest}}) = \text{RealConsMat}(j)$
11: $\quad$ Interference cancellation on $\text{Imaglayer}_{\text{weakest}}$-th layer
12: $\quad$ Normal V-BLAST algorithm on the remaining layers
13: **end for**
14: Combine 4 candidate symbol vectors as a matrix **ImagDetSym**
15: The candidates vector $\mathbf{DetSym} = [\mathbf{RealDetSym} \quad \mathbf{ImagDetSym}]$
16: $\text{BestSetIndex} = \min(\|\mathbf{y} - \mathbf{H} \times \mathbf{DetSym}\|^2)$
17: Compare $\text{DetSym}(\text{BestSetIndex})$ to **s**, calculate the number of symbol errors

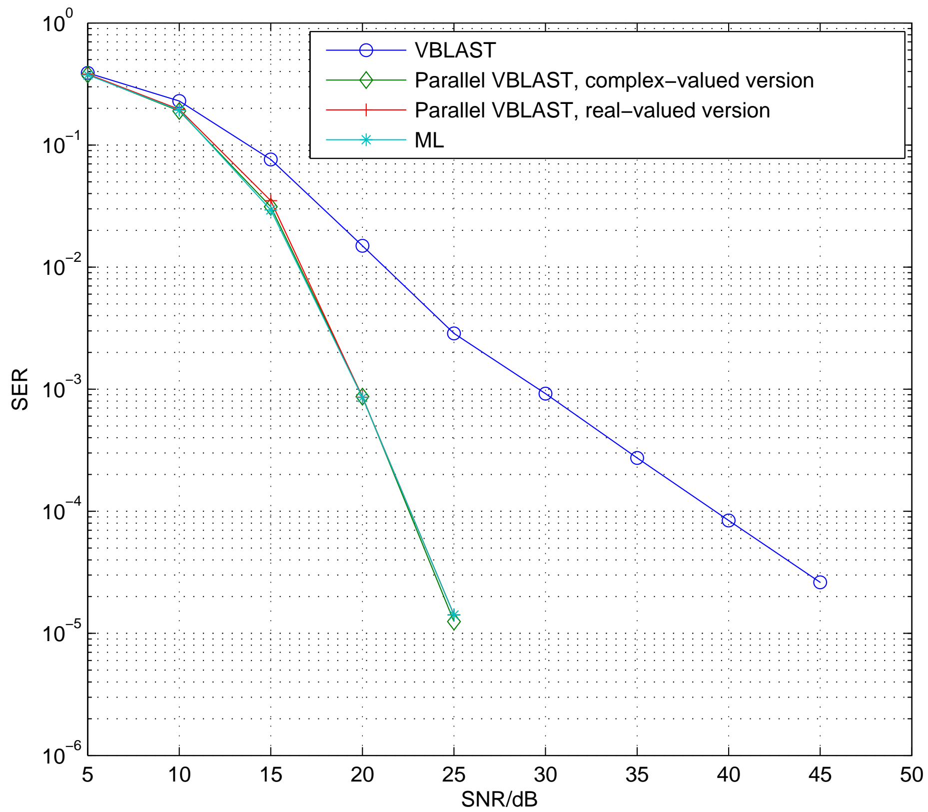

Figure 4.2: Performance of conventional V-BLAST, Parallel V-BLAST and Real-Imaginary component V-BLAST for a $M_t = M_r = 4$, 16-QAM MIMO system.

most reaches the optimal performance since the weakest channel layer is fully enumerated for all the possible candidates; then after the interference cancellation, the influence of the noise can be reduced as much as possible to ensure the accuracy of the detection on the rest of layers. The real and imaginary components version also performs near optimally.

4.4.5 The Parallel K-Best Algorithm

The algorithm considered in this section is the conventional K-Best algorithm that has been converted into a parallel version. As introduced in Chapter 2, the K-Best algorithm is a breadth-first sphere detector where the width of the search at each level in the tree is restricted to K. All the operations at the same level of the tree search can be transferred to the GPU and executed in parallel.

The structure of this algorithm is essentially the same as that of the K-Best algorithm which was introduced in Section 2.6.3. Recall that conditional statements are not allowed inside a gfor loop, but this restriction can be overcome by expressing the condition as a multiplied condition factor. (See Fig. 3.3 in Chpater 3)

The performance of the resulting parallel K-Best algorithm is shown in Fig. 4.3. With the increasing of the number of K, the performance of the parallel K-Best becomes better. When $K = 16$, the performance gives ML results.

4.4.6 The Fully Enumerated K-Best Algorithm

Since the performance of parallel V-BLAST shows a great reduction in SER compared to the conventional V-BLAST detector, we tried to apply the same strategy to the K-Best algorithm to see if improved performance would result. As was described above, the first step in designing a parallel V-BLAST algorithm was to find the weakest channel layer and then do a fully-enumerated breadth-first search of the original detector applied to the remaining layers. Modifying the original K-Best algorithm in the same way, the symbol vector is first separated into real and imaginary components and reshaped again as one real-valued symbol vector according to Eq. (2.6) in Chapter 2. So we only need to consider 4 possible values

(-3, -1, 1, 3) for the both real and imaginary components on this weakest layer. Then the normal K-Best procedure is executed on the remaining layers. At the end of this algorithm, there will be $2 \times 4 \times K$ candidate symbol vectors left. From these candidate solutions we pick the symbol vector that minimizes the predicted error metric.

The algorithmic procedure is provided in Algorithm 5. Most of the parameters in this algorithm are similar to those in Algorithm 4. Note that in the conventional K-Best algorithm, the strongest layer of the symbol vector is detected first. In the modified K-Best algorithm, the weakest layer is fully enumerated, and the remaining sub-trees are searched in K-Best fashion, with a total of K nodes expanded at each level.

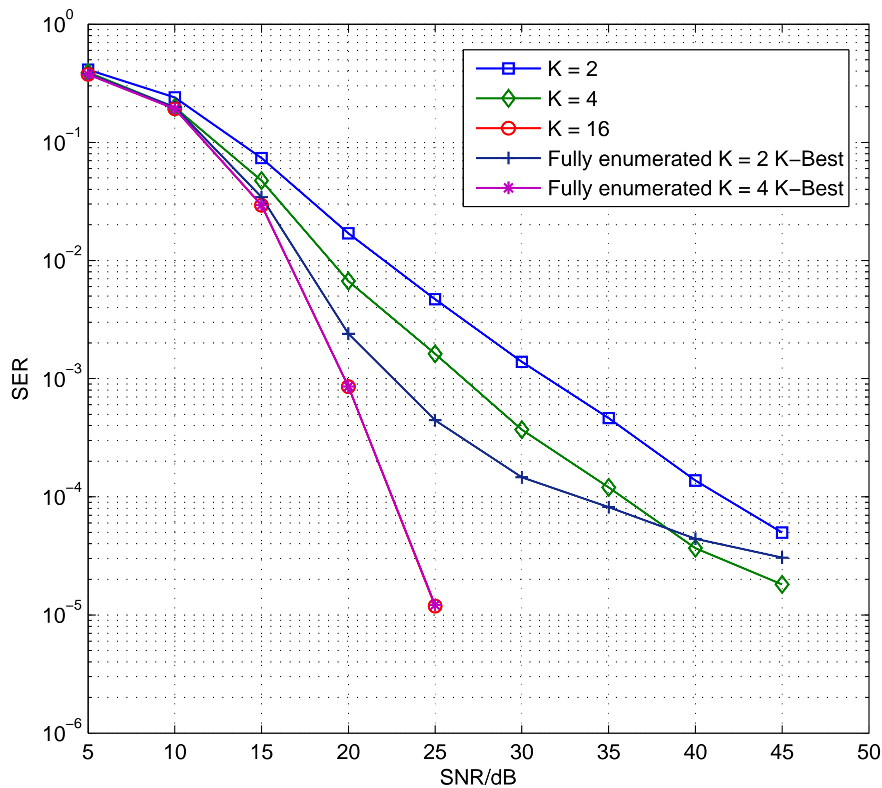

Figure 4.3: Performance of the K-Best and the fully enumerated K-Best for a $M_t = M_r = 4$, 16-QAM MIMO system.

Algorithm 5 The Fully Enumerated K-Best Algorithm

Inputs:

The numbers of transmitter and receiver's antennas, M_t and M_r, respectively;

16-QAM Constellation set **RealConsMat** and its size RealConsSize;

The $2M_r \times 2M_t$ real-valued channel matrix **H**, the real-valued symbol vector **s** and the real-valued received signal vector **y**;

The number K of selected best nodes on each layer;

The real-valued Moore-Penrose pseudo inverse matrix **G**;

Output:

The number of symbol errors from the detector;

1: $\text{Reallayer}_{\text{weakest}} = \max(\text{norm}(\mathbf{G}))$
2: $\text{Imaglayer}_{\text{weakest}} = \text{Reallayer}_{\text{weakest}} + \text{RealConsSize}$
3: Set $\text{Reallayer}_{\text{weakest}}$ as the first detected symbol layer
4: Reorder **H** as $\text{Reallayer}_{\text{weakest}}$ is the last channel layer
5: QR decomposition on the new ordered channel
6: **for all** $i = 1$: RealConsSize **do**
7: The first detected symbol = RealConsMat(i)
8: Normal K-Best algorithm on the remaining layers
9: **end for**
10: Combine four candidate symbol vectors as a matrix **RealDetSym**
11: Set $\text{Imaglayer}_{\text{weakest}}$ as the first detected symbol layer
12: Reorder **H** as $\text{Imaglayer}_{\text{weakest}}$ is the last channel layer
13: QR decomposition on the new ordered channel
14: **for all** $j = 1$: RealConsSize **do**
15: The first detected symbol = RealConsMat(j)
16: Normal K-Best algorithm on the remaining layers
17: **end for**
18: Combine four candidate symbol vectors as a matrix **ImagDetSym**
19: The candidates vector **DetSym** $=$ [**RealDetSym** **RealDetSym**]
20: BestSetIndex $= \min(\|\mathbf{y} - \mathbf{H} \times \mathbf{DetSym}\|^2)$
21: Compare DetSym(BestSetIndex) to **s**, calculate the number of symbol errors

The performance of this fully enumerated K-Best algorithm is illustrated in Fig. 4.3 for $K = 2$ and 4. The plots in Fig. 4.3 show that exhaustive enumeration over the weakest layer achieves good performance. Note that conventional 16-Best and 4-Best with full enumeration both approach the optimal detection curve. From this figure, we also see that 2-Best with full enumeration performs much better than conventional 4-Best in low SNR environments, but it approaches the performance of conventional 2-Best when the SNR becomes higher. In a higher SNR environment, the influence of the noise becomes more and more weak and detection errors are determined increasingly by the effects of interference and error propagation. In conclusion, the fully enumerated method produces more benefits in a lower SNR environment and it can also help to improve the detection accuracy of cheaper but less accurate detection algorithms.

4.4.7 The Parallel V-BLAST with K-Best Algorithm

Compared to the K-Best algorithm, the complexity of V-BLAST is lower since it only needs to do the quantization and interference cancellation steps at each level to get a detected symbol vector. However, lower detection accuracy is a price of lower complexity. As we explained and demonstrated above, the parallel V-BLAST algorithm with real and imaginary components provides near-optimal performance. But the inevitable cost is extra channel reordering 8 times after each interference cancellation step. The channel matrix inversion is one step in the reordering, which is also the bottleneck of the acceleration if we are using the Jacket library in MATLAB. We thus decided to combine parallel V-BLAST and the K-Best algorithm to minimize the number of matrix inverse calculations. Since in the K-Best algorithm, the last symbol of the symbol vector is detected first, we can apply the real and imaginary parallel V-BLAST detector first, then we can run the K-Best algorithm running on the remaining layers to finish the detection. The best division of layers between parallel V-BLAST and K-Best is to be determined. After this processing, there are $8 \times K$ candidate symbol vectors, according to 16-QAM, and we must pick the best one as the detected symbol vector as before. In this algorithm, our purpose is to reduce the computing complexity by cutting down on the usage of K-Best. The

brief structure of this algorithm is shown in Fig. 4.4.

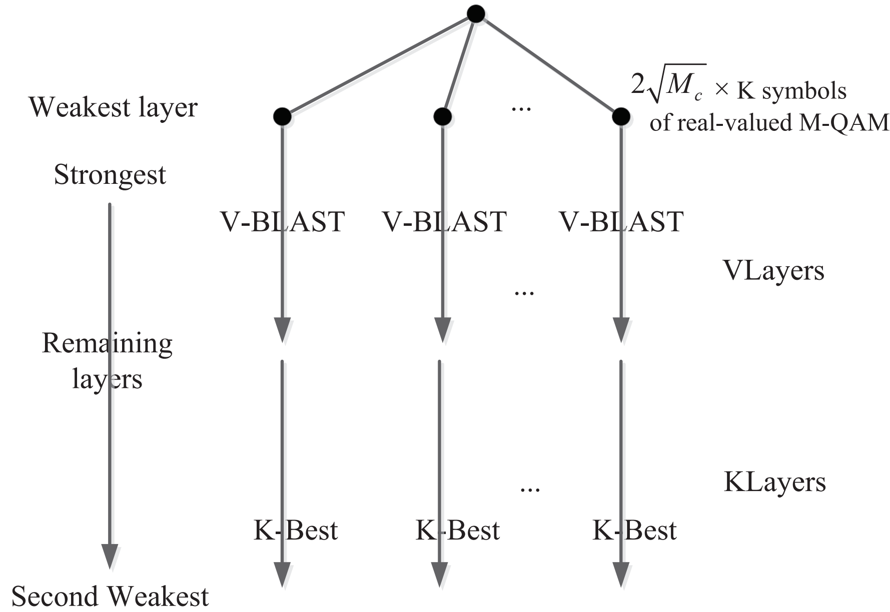

Figure 4.4: Algorithmic structure of the parallel V-BLAST with K-Best algorithm.

Algorithm 6 shows the procedure of this algorithm. The main strategy of Algorithm 6 is still to fully enumerate the weakest symbol layer, then the remaining layers are detected by both the V-BLAST and K-Best algorithms. Parameters $KLayer$ and $VLayer$ can be chosen from 1 to $2 * M_t - 2$. Note that the sum of $KLayer$ and $VLayer$ is always $2M_t - 1$.

The resulting performance can be seen in Fig. 4.5. The conventional K-Best algorithm (for $K = 2, 4and16$) was compared to two extreme versions of the new parallel algorithm: one executes parallel V-BLAST on the first 6 layers and 2-Best algorithm on the last layer. For this version, we can see from Fig. 4.5 that the performance is better than that of the normal 4-Best algorithm. But at the other extreme, if only one layer is detected by parallel V-BLAST and the rest are detected using 2-Best, the performance is almost the same as the conventional serial 4-Best algorithm. The conclusion is that parallel V-BLAST performs better than parallel V-BLAST with the last layer using the 2-Best algorithm.

~

Algorithm 6 The parallel V-BLAST with K-Best algorithm

Inputs:

The numbers of transmitter and receiver antennas, M_t and M_r, respectively;

16-QAM Constellation set **RealConsMat** and its size RealConsSize;

The $2M_r \times 2M_t$ real-valued channel matrix **H**, the real-valued symbol vector **s** and the real-valued received signal vector **y**;

The number K of selected best nodes on each layer;

The number $KLayer$ of layers that apply the K-Best algorithm;

The number $VLayer$ of layers that apply the V-BLAST algorithm;

The real-valued Moore-Penrose pseudo inverse matrix **G**;

Output:

The number of symbol errors from the detector;

1: $\text{Reallayer}_{\text{weakest}} = \max(\text{norm}(\mathbf{G}))$
2: $\text{Imaglayer}_{\text{weakest}} = \text{Reallayer}_{\text{weakest}} + \text{RealConsSize}$
3: **for all** $i = 1 : \text{RealConsSize}$ **do**
4:     $\text{RealDetSym}(\text{Reallayer}_{\text{weakest}}) = \text{RealConsMat}(i)$
5:     Interference cancellation on $\text{Reallayer}_{\text{weakest}}$-th layer
6:     Normal V-BLAST algorithm on the remaining $2 : VLayer$ layers
7: **end for**
8: Reorder **H**, set the last $KLayer$ channel layers as the pending layers
9: QR decomposition on the new ordered channel
10: Normal K-Best algorithm on the remaining layers
11: Get detected symbol vector **RealDetSym**
12: **for all** $i = 1 : \text{RealConsSize}$ **do**
13:     $\text{ImagDetSym}(\text{Imaglayer}_{\text{weakest}}) = \text{RealConsMat}(i)$
14:     Interference cancellation on $\text{Reallayer}_{\text{weakest}}$-th layer
15:     Normal V-BLAST algorithm on the remaining $2 : VLayer$ layers
16: **end for**
17: Reorder **H**, set the last $KLayer$ channel layers as the pending layers
18: QR decomposition on the new ordered channel
19: Normal K-Best algorithm on the remaining layers
20: Get detected symbol vector **ImagDetSym**
21: The candidates vector $\mathbf{DetSym} = [\mathbf{RealDetSym} \quad \mathbf{RealDetSym}]$
22: $\text{BestSetIndex} = \min(\|\mathbf{y} - \mathbf{H} \times \mathbf{DetSym}\|^2)$
23: Compare DetSym(BestSetIndex) to **s**, calculate the number of symbol errors

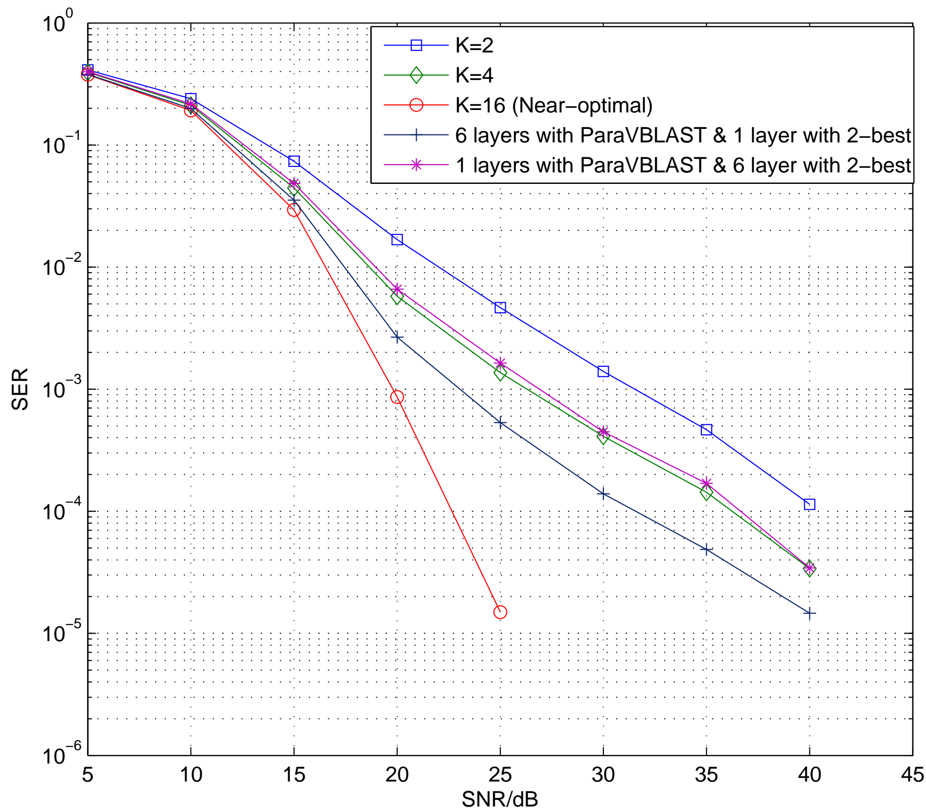

Figure 4.5: Performance of the VBLAST-KBest hybrid MIMO detector for a $M_t = M_r = 4$, 16-QAM MIMO system.

Chapter 5

Parallel Implementation of MIMO Detection Algorithms Using the Parallel Computing Toolbox in MATLAB

As was described in Chapter 3, there are several ways to implement parallelism in computation. In Chapter 4, we investigated the benefits of GPU programming for the parallel simulation of MIMO detectors. We initially chose the Jacket library as the software framework to accelerate those different MIMO detection algorithms. The results showed that the achieved of acceleration is limited by the relatively poor performance of matrix multiplications. In order to investigate the alternative of multicore parallelism, we used the parallel computing toolbox (PCT) in MATLAB to implement MIMO detectors.

5.1 Parallelism in MATLAB

MATLAB is a widely used tool with a large collection of built-in functions. Multithreading is a system capability that MATLAB already exploits by default to speed up many of the built-in functions. This is perhaps one of the reasons why our GPU computing experiments have not produces competitive speed-ups when the size of the data structures is not big enough.

Nowadays, most CPUs in desktop computers, laptops, tablets and even cellphones are multicore. The MATLAB parallel computing toolbox (PCT), which is intended to exploit multicore CPUs and clusters of computers, is another parallel strategy that we investigated in this research.

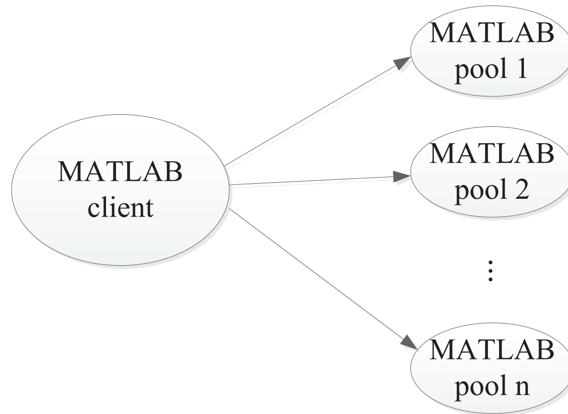

Figure 5.1: MATLAB parallel computing toolbox worker pool structure

Fig. 5.1 briefly shows that how a program is parallelized with the PCT. The MATLAB client stands for the copy of the MATLAB that we start in the regular way. The MATLAB pool, which is also called as “worker” in some of the documents, is the the copy that are created to help in the computation. The pool can be seen as a “lab” in MATLAB, where the lab is the space that the data will be distributed to. Each of the lab can either be independent with each other or communicate if necessary. The number of labs depends on the number of cores on one or multiple workstations.

The PCT starts the parallelism by opening multiple labs in MATLAB. On the local computer we must enter:

- matlabpool open local #
- matlabpool close

The command “matlabpool open local # ” creates a pool of parallel MATLAB threads. The sign # stands for the number of labs that we need to open to form the pool. When all the parallel programs are finished, the labs can be shut down

by the second command “matlabpool close”. Since the overhead of opening the MATLAB labs is relatively expensive, we should make sure that all the parallel computing should be finished before we close the pool of labs.

PCT is easy to apply in our programs since most of the built-in functions in MATLAB are almost multithreading aware. Only relatively small changes are required to our program, mainly for some commands that are related to the parallelism.

Similar to the gfor-loop structure in Jacket for the GPU, the parfor-loop structure in PCT can replace a conventional for-loop to provide parallel computation. Instead of being executed in serial, the commands in the parfor-loop are executed in parallel. The total number of iterations is automatically distributed over the number of labs that are open. Each group of iterations will be executed at the same time. The computation within each iteration of the loop should be independent of all the others. The parallelism pattern of the parfor structure is the task parallel and there is no communication between the labs.

PCT also provides another command “spmd”, which stands for the “single program, multiple data”. This command can automatically distribute a large array over parallel hardware by dividing it into pieces for each of the lab in MATLAB. The parallelism pattern of the SPMD structure is the data parallel and the parallel labs can communicate with each other under the SPMD model.

In our research, we deal with million element data sets with one algorithm at a time, where each iteration of the loop is independent of the others, so we decided to apply the parfor structure to implement parallelism.

5.2 Matrix Multiplication Using the Parallel Computing Toolbox

In Chapter 4, the acceleration of matrix multiplication on the GPU is not remarkable when the matrix is small. In this Chapter, we investigate the problem of making direct and full use of the parallel CPU cores to see how much speed-up can be achieved.

Code Listing 5.1: Matrix Multiplication Benchmark using the parfor loop

```
matlabpool open NumPool
parfor ii = 1:102400
    A = rand(N,N);
    B = rand(N,N);
    C(:,:,ii) = A*B;
end
matlabpool close
```

The matrix multiplication benchmark in Code Listing 5.1 is similar to the Code Listing 4.2. The only change is that the parfor structure is used instead of for.

As in Chapter 4, we get a comparison among the for, gfor and parfor loops. The results are shown in Table 5.1.

Table 5.1: Matrix multiplication times (in seconds) using the for, gfor and parfor loops

Size of Matrix	Serial for-loop	gfor-loop on GPU	parfor-loop with number of pools				
			1	2	3	4	8
4	0.35	0.33	0.42	0.24	0.18	0.15	0.14
8	0.54	0.33	0.71	0.39	0.31	0.25	0.24
16	1.17	0.34	1.58	0.91	0.73	0.64	0.62
32	3.62	0.35	5.25	3.20	2.54	2.38	2.36
64	14.71	0.38	-	14.22	11.17	10.64	10.46

In Table 5.1, the running time includes the total number of iterations which is 102400. The degree of the parallelism depends on the number $NumPool$ of the MATLAB pools that have been opened during the computation. When using the parfor-loop, MATLAB distributes the 102400 iterations into $NumPool$ groups. For each group, MATLAB serializes the data first and then execute all the commands in the parfor-loop. This is why the results from the parfor-loop are even larger than the serial version when the number of pools is 1. This is also the reason why the

data is not available when the size of matrix is 64 and the number of open pools in MATLAB is 1. The resulting error from MATLAB is “Attempt to serialize data which is too large”.

The results from the for-loop and the gfor-loop are a little different from Table 4.2. This is because we only consider the multiplication alone in each iteration in the previous test results, as obtained from the profiler in MATLAB. In this test we include the time for random number generation as well, and use the tic-toc functions in the programs to accurately determine the running time. It can be seen that when the matrix size is small, multiplication time with gfor-loop on the 1024-core GPU is almost the same as for the serial for-loop, while the performance of the parfor-loop becomes better when the number of open labs increases. But when the matrix size grows, the running times for both the for and parfor loops are influenced a lot while the running times for the gfor loop stay almost the same.

5.3 Parallelism Models and the Performance Achieved Using the Parallel Computing Toolbox

Algorithm 7 Parallelism Model Using PCT

1: matlabpool open local $NumPool$
2: **parfor all** $loop = 1 : NumLoops$ **do**
3: $M_t \times 1$ symbol vectors are generated
4: $M_r \times M_t$ channel matrices are generated
(10 symbol vectors are processed for each new channel)
5: $M_r \times 1$ Gaussian noise samples are generated
6: Detection algorithm is applied on $NumPool$ symbols in parallel
7: $NumPool$ symbols are detected in parallel
8: **end for**
9: matlabpool close

In Algorithm 7, $NumPool$ stands for the number of labs that we decide to open. The degree of parallelism is determined by $NumPool$. $NumLoops$ is the number of outside loops to repeat the same algorithm. M_t and M_r are still the number of

antennas at the transmitter and the receiver.

Then each of the detection algorithms which were described in Chapter 2 and Chapter 4 can be substituted for step 6. A serial version of the algorithms can be used with rarely changing in Algorithm 7, the only change is apply the parfor-loop instead of for-loop. The communication system environment is set to be the same as in Chapter 4.

In order to have a clear view of acceleration using the different methods, the complete form of the running times comparison of different MIMO detection algorithms using serial and all kinds of parallel versions is provided in Table 5.2.

Table 5.2 shows the comparison among the serial version, the parallel version using Jacket on the GPU, and the parallel versions using PCT on the CPU. The data in first seven columns are the same as those in Table 4.5 in Chapter 4; the last four columns include the new results for the PCT version. The numbers of labs are set to be 4 and 8 in this test.

It can be seen from each of the speed-up columns that the acceleration for all detection algorithms are similar, they are affected by the number of open labs. For the ZF and MMSE algorithms, since the matrix multiplications are only applied a few times, the advantage of GPU computing is more than the PCT. In the V-BLAST algorithm, matrix multiplication is frequently used, and so PCT performance is better than the Jacket performance. The K-Best algorithm was described in Chapter 2, we can see from the procedures that matrix multiplication is used intensively. When the size of the matrix is only 4, the Jacket does not provide much acceleration, while the PCT can still distribute the data and the instructions to all 4 labs to reduce the calculation time, which also shows the advantages of task parallelism.

For all these detection algorithms, the speed-up factors are stay almost around 4 with when the number of open labs is fixed at 4. When the number of open labs increases to 8, we can get some benefits from 4 more labs but not too much since the physical cores of our PC is 4. With the help of multithreading technique in 4 cores, 8 threads are available for the calculation.

The parallel V-BLAST algorithm has the same strategy as the conventional serial V-BLAST, except the weakest channel layer is fully enumerated, which makes

Table 5.2: Running times (in seconds) comparison of MIMO detection algorithms with the serial and different parallel versions

	Serial 1024000	GPU gfor $1000 * 1024$	Speed Up	GPU gfor $F = 18$ $1000 * 1024$	Speed Up	GPU gfor $10 * 102400$	Speed Up	PCT with 4 workers 1024000	Speed Up	PCT with 8 workers 1024000	Speed Up
Channel Generation	29.574	0.107	276.393	0.311	95.093	0.017	1770.900	1.819	16.256	1.489	19.858
Data Generation	39.879	0.670	59.521	1.086	36.721	0.172	231.855	7.083	5.630	4.563	8.740
ZF Detection	278.603	15.282	18.231	6.552	42.522	8.030	34.695	71.591	3.892	54.455	5.116
MMSE Detection	287.134	15.296	18.772	6.572	43.691	8.044	35.695	72.399	3.966	56.031	5.125
V-BLAST Detection	626.020	191.333	3.272	133.290	4.697	64.530	9.701	158.114	3.959	116.038	5.395
K-Best Detection	703.008	620.127	1.134	-	-	490.303	1.434	204.663	3.435	209.069	3.363
Parallel V-BLAST Detection	3075.281	513.076	5.994	234.749	13.100	290.964	10.569	932.596	3.298	677.297	4.541

the calculation times larger than for the normal V-BLAST algorithm. So we can see that, under the condition of large computation, GPU can achieve more speed-up than the CPU.

$\sim$

Chapter 6

Conclusions

6.1 Contributions

In Chapter 2, we briefly reviewed the fundamentals of MIMO wireless technology and described the major classes of detection algorithms. The detectors included three linear-complexity algorithms (ZF, MMSE, V-BLAST), and then the more complex, but more accurate, sphere detection algorithms (FP, SE, K-Best). Note that the FP and SE sphere detection algorithms are depth-first and can not easily be parallelized because each branch of the search tree would be different if different data is applied, and it is awkward and usually inefficient to execute different codes simultaneously.

Chapter 3 introduced various ways to exploit hardware parallelism such as using FPGA, custom VLSI and GPU technology. We briefly reviewed the architecture of GPU units and described the various kinds of memories allocated inside these units. The GPU is not only used in the graphics processing field, but it can also be used for general-purpose computing. General-purpose computing on the GPU has recently received a lot of attention because of the potential benefits of significant and relatively cheap speed-up. Access to the GPU can be achieved using a variety of programming environments such as CUDA, OpenCL and Jacket. Our implementations used Jacket and the parfor construct in the MATLAB parallel computing toolbox. In addition, we reviewed the literature to see what other researchers have achieved in this area.

The main focus of this thesis is the implementation of the MIMO detection algorithm on the GPU, as described in Chapter 4. We chose the Jacket function library because of its compatibility with MATLAB. First, we did experiments on matrix-vector multiplication benchmarks using the GPU to find out how much improvement we could expect to achieve from the parallelism. The disappointing result was that the large reported speed-ups for other matrix-oriented problems on the GPU only seem to be attainable with relatively large matrices. So in Experiment 3, we tried to merge several small matrices into the diagonal of one large matrix, and then executed the multiplication by the large matrix, the results showed that we can take more benefits from the GPU when the size of matrix is larger. Our existing detection code was already designed in MATLAB, and it would be easier and clearer to compare (serial and parallel versions) of alternative detectors if we used the same programming environment. In data generation, we readily achieved significant speed-ups on the GPU. When parallelizing the algorithms, we had to pay much attention to the restrictions of Jacket library, and make sure that all the data structures were processed as much as possible locally on the GPU. We also proposed a new MIMO detection algorithm called "Parallel VBLAST-KBest". This algorithm combined the strategies of Parallel V-BLAST and K-Best together to reduce the computational complexity of V-BLAST and increase the accuracy of normal K-Best.

In Chapter 5, we investigated another way to implement the parallelism. Since the matrix multiplication was still the problem sometimes in Chapter 4, we tried to apply the parallel computing toolbox in MATLAB to achieve acceleration by taking advantage of multiple cores on the CPU. Also, we repeated the same experiments on the basic MIMO detection algorithms to see how much improvement we can get from multicore computer parallelism compared to GPU programming.

6.2 Future Work

During this research project, we applied the simple parallelism model as described in Chapter 4, where 1024 different data streams were processed with the same op-

erations simultaneously on 1024 GPU cores. In Chapter 5, we described a brief investigation about the multicore CPU parallelism using parallel computing toolbox in MATLAB. However, there are many models of parallelism. One strategy is to divide the multiple data streams into several different groups that can execute different commands at the same time. While the execution threads in each group are in parallel, after a certain period of time, the results from different groups can be combined into the same data structure and the rest of the calculation can be completed serially.

All of the parallel MIMO detectors that were investigated in this thesis were implemented using both the Jacket function library with GPU and the parallel computing toolbox in MATLAB with multicore CPU. It is possible that the relatively high-level data structures and functions are limited compared to the lower-level GPU language CUDA C/C++. If all or even just the critical parts of the programs could be re-implemented in CUDA C/C++ and called in MATLAB, the performance of the parallel MIMO detection algorithms might be found to be greatly improved.

~

Bibliography

[1] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013. [Online]. Available: http://books.google.com/books?id=ynydqKP225EC

[2] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Inf. Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[3] J. L. Hennessy, A. C. Arpaci-Dusseau, and D. A. Patterson, *Computer architecture : a quantitative approach*. Amsterdam ; Boston : Elsevier/Morgan Kaufmann Publishers, c2007., 2007.

[4] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, 1948.

[5] "IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control (MAC)and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput," *IEEE Std 802.11n-2009 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, and IEEE Std 802.11w-2009)*, pp. 1 –565, 29 2009.

[6] I. T. U.-R. Bureau, "ITU global standard for international mobile telecommunications IMT-Advanced," ITU-R, March 2008. [Online]. Available: http://wirelessman.org/liaison/docs/L80216-08_008.pdf

[7] G. T. S. Group, "Spatial channel model, SCM-134 text V6.0," in *Spatial Channel Model AHG (Combined as-hoc from 3GPP and 3GPP2)*, April 2003.

[8] C. Eklund, R. Marks, K. Stanwood, and S. Wang, "IEEE standard 802.16: a technical overview of the WirelessMAN/sup TM/ air interface for broadband wireless access," *IEEE Commun. Mag.*, vol. 40, no. 6, pp. 98 –107, june 2002.

[9] G. J. Foschini, "Layered space-time architecture for wireless communication in a fading environment when using multi-element antennas," *Bell Labs Tech. J.*, vol. 1, pp. 41–59, Summer 1996.

[10] V. Tarokh, N. Seshadri, and A. Calderbank, "Space-time codes for high data rate wireless communication: performance criterion and code construction," *IEEE Trans. Inf. Theory*, vol. 44, no. 2, pp. 744–765, 1998.

[11] G. Stuber, J. Barry, S. McLaughlin, Y. Li, M. Ingram, and T. Pratt, "Broadband MIMO-OFDM wireless communications," *Proc. IEEE*, vol. 92, no. 2, pp. 271 – 294, feb 2004.

[12] G. Frank, "Pulse code communication," U.S. Patent US2 632 058 A, March 17, 1953.

[13] L. Zheng and D. Tse, "Diversity and multiplexing: a fundamental tradeoff in multiple-antenna channels," *IEEE Trans. Inf. Theory*, vol. 49, no. 5, pp. 1073–1096, 2003.

[14] O. Damen, A. Chkeif, and J.-C. Belfiore, "Lattice code decoder for space-time codes," *IEEE Commun. Lett.*, vol. 4, no. 5, pp. 161–163, 2000.

[15] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger, "Closest point search in lattices," *IEEE Trans. Inf. Theory*, vol. 48, no. 8, pp. 2201–2214, 2002.

[16] E. H. Moore, "On the reciprocal of the general algebraic matrix," *B. Am. Math. Society*, vol. 26, pp. 394–395, 1920.

[17] R. Penrose, "A generalized inverse for matrices," *Math. Proc. Cambridge*, vol. 51, no. 03, pp. 406–413, 1955. [Online]. Available: http://dx.doi.org/10.1017/s0305004100030401

[18] A. Alimohammad, S. Fard, and B. Cockburn, "Improved layered MIMO detection algorithm with near-optimal performance," *Electron. Lett*, vol. 45, no. 13, pp. 675–677, 2009.

[19] G. H. Golub and C. F. Van Loan, *Matrix computations (3rd ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[20] S. Even, *Graph algorithms*. Cambridge University Press, New York, 2012.

[21] B. Hassibi and H. Vikalo, "On the sphere-decoding algorithm I. Expected complexity," *IEEE Trans. Signal Process.*, vol. 53, no. 8, pp. 2806–2818, 2005.

[22] C. P. Schnorr and M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems," *Math. Program.*, vol. 66, pp. 181–199, 1994, 10.1007/BF01581144. [Online]. Available: http://dx.doi.org/10.1007/BF01581144

[23] Z. Guo and P. Nilsson, "Reduced complexity Schnorr-Euchner decoding algorithms for MIMO systems," *IEEE Commun. Lett.*, vol. 8, no. 5, pp. 286–288, 2004.

[24] ——, "Algorithm and implementation of the K-best sphere decoding for MIMO detection," *IEEE J. Sel. Areas Commun.*, vol. 24, no. 3, pp. 491 – 503, march 2006.

[25] M. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, no. 12, pp. 1901 – 1909, dec. 1966.

[26] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput.*, vol. C-21, no. 9, pp. 948 –960, Sept. 1972.

[27] B. Barney. Introduction to parallel computing. Lawrence Livermore National Laboratory. UCRL-MI-133316. [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/

[28] C. G. Bell, W. Broadley, W. Wulf, A. Newell, C. Pierson, R. Reddy, and S. Rege, *C.mmp: the cmu multiminiprocessor computer - requirements and overview of the initial design*, ser. Research paper. Department of Computer Science, Carnegie-Mellon University, 1971, vol. 71. [Online]. Available: http://books.google.ca/books?id=iXprNwAACAAJ

[29] W. A. Wulf and C. G. Bell, "C.mmp: a multi-mini-processor," in *Proceedings of the December 5-7, 1972, fall joint computer conference, part II*, ser. AFIPS '72 (Fall, part II). New York, NY, USA: ACM, 1972, pp. 765–777. [Online]. Available: http://doi.acm.org/10.1145/1480083.1480098

[30] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[31] N. Corporation. The World's First GPU. NVIDIA Corporation. [Online]. Available: http://www.nvidia.com/page/geforce256.html

[32] A. LLC. Jacket Documentation. AccelerEyes LLC. [Online]. Available: http://wiki.accelereyes.com/wiki/index.php/GFOR_Usage

[33] Y. Zhang and K. Parhi, "Parallel Architecture of List Sphere Decoders," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, 2007, pp. 2096–2099.

[34] W. Hongyuan and C. Muyi, "A Fixed-Complexity Sphere Decoder for MIMO Systems on Graphics Processing Units," in *Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on*, 2010, pp. 1–4.

[35] P. Cervantes-Lozano, L. Gonzalez-Perez, and A. Garcia-Garcia, “Analysis of Parallel Sorting Algorithms in K-best Sphere-Decoder Architectures for MIMO Systems,” in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, 2011, pp. 321–326.

[36] J. Yu, J. Ma, and Z. Mao, “Parallel SFSD MIMO detection with SOFT-HARD combination enumeration,” in *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, 2011, pp. 228–233.

[37] S. Roger, C. Ramiro, A. Gonzalez, V. Almenar, and A. Vidal, “Fully Parallel GPU Implementation of a Fixed-Complexity Soft-Output MIMO Detector,” *IEEE Trans. Veh. Technol.*, vol. 61, no. 8, pp. 3796–3800, 2012.

[38] H.-W. Liang, W.-H. Chung, H. Zhang, and S.-Y. Kuo, “A parallel processing algorithm for Schnorr-Euchner sphere decoder,” in *Wireless Communications and Networking Conference (WCNC), 2012 IEEE*, 2012, pp. 613–617.

[39] M. Harris, “Mapping computational concepts to GPUs,” in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH ’05. New York, NY, USA: ACM, 2005, pp. 50+. [Online]. Available: http://doi.acm.org/10.1145/1198555.1198768

[40] D. Luebke and G. Humphreys, “How GPUs Work,” *Computer*, vol. 40, no. 2, pp. 96–100, Feb. 2007. [Online]. Available: http://dx.doi.org/10.1109/MC.2007.59

[41] A. F. Pankeu Yomi, “A Near-Optimal and Efficiently Parallelizable Detector for Multiple-Input Multiple-Output Wireless Systems,” Master’s thesis, University of Alberta, 2012.

~

Appendix A

Source Codes for Serial MIMO Detection Algorithms

A.1 Main Function for Different Detection Algorithms

```
M_transmit = 4;
M_receive = 4;
m_dimension = 2*M_transmit;  % Channel layer
M_QAM = 16;
Energy = 2*(M_QAM-1)/3;
SNR = 5:5:30;

NumSymbs = 1024000;
NumSymbsPerChan = 10;

Constellation_point = [-3 -1 1 3];    % 16-QAM
lengthConstellation = sqrt(M_QAM);
partition = zeros(1,lengthConstellation-1);
for p = 1:lengthConstellation-1)
    partition(p) = (Constellation_point(p) + Constellation_point
      (p+1))/2;
end
% Complex-vlaued Constellation
ConsCount = 1;
ConsMat = zeros(1,M_QAM);
for row = 1:sqrt(M_QAM)
    for col = 1:sqrt(M_QAM)
        ConsMat(ConsCount) = Constellation_point(row) + 1i*
```

```
          Constellation_point(col);
         ConsCount = ConsCount + 1;
    end
end

for i = 1:length(SNR)
    snr = 10^(SNR(i)/10);
    ChanAge = NumSymbsPerChan;
    Error_temp = zeros(1,NumSymbs);
    for ss = 1:NumSymbs
        % Generate the transmit signal,channel, noise
        x = randint(M_transmit,1,[0,M_QAM-1]);
        s = qammod(x,M_QAM,0,'Gray');

        noise = complex(randn(M_receive,1),randn(M_receive,1))*
         sqrt(Energy*M_transmit/(2*snr));

        ChanAge = ChanAge + 1;
        if (ChanAge > NumSymbsPerChan)
                H = complex(randn(M_receive,M_transmit),randn(
                 M_receive,M_transmit))/sqrt(2);
        end

        % Received signal
        y = H*s + noise;

        % Detection algorithm

        % Calculate the symbol error rate
        Error_temp(ss) = Error_temp(ss)+error/(2*M_transmit);
    end
    SER(i) = sum(Error_temp)/NumSymbs;
end
```

A.2 Maximum Likelihood (ML) Detection Algorithm

```
function [s_det,error] = ML_Dete(y,H,s,M_QAM,ConstMat)

% Input:  y: complex-valued received signal
%         H: complex-valued channel matrix
%         s: complex-valued transmitted signal
```

```
%         M_QAM: size of modulation scheme
%         ConstMat: complex-valued constellation
% Output: s_det: detected symbol vector
%         error: number of erroneously detected symbols

SymbolSize = length(s);
s_det = zeros(SymbolSize,1);
error = 0;
temps = zeros(SymbolSize,1);
s_det(:,1) = ConstMat(1);
min_value = norm(y- H*s_det)^2;
for dd = 1:M_QAM
    layer = 1;
    temps(layer) = ConstMatrix(dd);
    for cc = 1:M_QAM
        layer = 2;
        temps(layer) = ConstMatrix(cc);
        for bb = 1:M_QAM
            layer = 3;
            temps(layer) = ConstMatrix(bb);
            for aa = 1:M_QAM
                layer = 4;
                temps(layer) = ConstMat(aa);
                temp_norm = norm(y - H*temps)^2;
                if temp_norm < min_value
                        s_det = temps;
                        min_value = temp_norm;
                end
            end
        end
    end
end
% Calculate the symbol errors
for ee = 1:SymbolSize
    RealCount = real(s_det(ee)-real(s(ee));
    Realcondition = RealCount~=0;
    error = error+Realcondition;
    ImgCount = imag(s_det(ee))-imag(s(ee));
    Imgcondition = ImgCount~=0;
    error = error+Imgcondition;
end
```

A.3 Zero Forcing (ZF) Detection Algorithm

```
function error = ZF(s,H,noise,Constellation_point,partition)

% Input:  s: complex-valued transmitted signal
%         H: complex-valued channel matrix
%         noise: complex-Valued noise
%         Constellation_point: real-valued constellation
%         partition: constellation points partition
% Output: error: number of erroneously detected symbols

Q = (H'*H)\H';
error = 0;
for i = 1:length(s)
    Y(:,i) = H(:,i)*s(i)+noise;
    % Nulling and Slicing
    [~,R] = quantiz(real(Q(i,:)*Y(:,i)),partition,
     Constellation_point);
    [~,Img]= quantiz(imag(Q(i,:)*Y(:,i)),partition,
     Constellation_point);
        % Symbol Errors
    if R~=real(s(i))
        error = error+1;
    end
    if Img~=imag(s(i))
        error = error+1;
    end
end
```

A.4 Minimum Mean Square Error (MMSE) Detection Algorithm

```
function error = MMSE(s,H,noise,Constellation_point,partition)

% Input:  s: complex-valued transmitted signal
%         H: complex-valued channel matrix
%         noise: complex-Valued noise
%         Constellation_point: real-valued constellation
%         partition: constellation points partition
% Output: error: number of erroneously detected symbols
```

```
Q = (H'*H+(1/snr)*eye(M_transmit,M_transmit))\H';
error = 0;
for i = 1:length(s)
    Y(:,i) = H(:,i)*s(i)+noise;
    % Nulling and Slicing
    [~,R] = quantiz(real(Q(i,:)*Y(:,i)),partition,
     Constellation_point);
    [~,Img]= quantiz(imag(Q(i,:)*Y(:,i)),partition,
     Constellation_point);
        % Symbol Errors
    if R~=real(s(i))
        error = error+1;
    end
    if Img~=imag(s(i))
        error = error+1;
    end
end
```

A.5 V-BLAST Detection Algorithm

```
function error = VBLAST(s,H,y,Constellation_point,partition)

% Input:  s: complex-valued transmitted signal
%         H: complex-valued channel matrix
%         y: complex-Valued received signal
%         Constellation_point: real-valued constellation
%         partition: constellation points partition
% Output: error: number of erroneously detected symbols

G = (H'*H+(1/snr)*eye(M_transmit,M_transmit))\H';
k = zeros(1,length(s));
s_det = zeros(1,length(s));
error = 0;
for i = 1:length(s)
        for p = 1:length(s)
                Q(p) = (norm(G(p,:)))^2;   % calculate the
                 normal value of G
        end
        for t = 1:i-1
                Q(k(t)) = Inf;     % set the detected normal
```

```
            value to infinity
        end
        [~,I] = min(Q);         % I stands for the subscript of
         the minimum value in the normal value set
        k(i) = I;               % save the subscript
        shk = G(I,:)*Y;         % nulling
        % slicing
        [~,R] = quantiz(real(shk),partition,Constellation_point)
         ;
        [~,Img] = quantiz(imag(shk),partition,
         Constellation_point);
        s_det(I) = R+1j*Img;
        Y = Y-s_det(I)*H(:,I); % interference cancellation
        H(:,I) = 0;             % set the used channel into 0
        G = pinv(H);            % peseudo inverse for the new
         channel
        % Symbol Errors
        if R~=real(s(I))
                error = error+1;
        end
        if Img~=imag(s(I))
                error = error+1;
        end
end
```

A.6 Fincke-Pohst (FP) Sphere Detection Algorithm

```
% Before the algorithm, System should be real-valued
S_r = [real(s);imag(s)];     % Generate the real version
H_r = [real(H) -1*imag(H);imag(H) real(H)];
noise_r = [real(noise);imag(noise)];
Y_r = H_r*S_r+noise_r;       % The real system
[Q,R] = qr(H_r);             % QR decomposition
for k = 1:length(Y_r)
    if (R(k,k)<0)
        Q(:,k) = Q(:,k)*(-1);
         R(k,:) = R(k,:)*(-1);
        end
end
Z_r = Q'*Y_r;
```

```
% Radius for FP-SD
variance2 = (M_transmit*Energy/(2*log2(M_QAM)))/snr; % variance
 of the noise
Probability2 = 0.01;
d = 2*chi2inv((1-Probability2),m_dimension)*variance2;

function [det_node,num_nodes,error] = FP_SD(m_dimension,R,d,Y_r,
 H_r,Z_r,S_r,Constellation_point)

% Input:  m_dimension: search level
%         R: the upper triangular from the QR factorization
%         d: the search radius
%         Y_r: real-valued received signal y
%         H_r: real-valued channel H
%         Z_r: real-valued Z
%         S_r: real-valued transmitted signal s
%         Constellation_point: real-valued constellation
% Output: det_node: the matrix of the detected node at each
 level
%         num_nodes: the number of the expanded nodes
%         error: number of erroneously detected symbols

a = inf;
num_nodes = 0;
k = m_dimension;            % search level
D(k) = d;   % the radius matrix
s = zeros(m_dimension,1);   % initialze the detected result
det_node = zeros(m_dimension,1);
error = 0;
while (k~=0)
    rs = 0;
    for t = (k+1):m_dimension
        rs = rs+R(k,t)*s(t); % Sumation of r*s
    end
    lower_bound(k) = (Z_r(k)-rs-sqrt(D(k)))/R(k,k);        % set
      the lower_bound
    upper_bound(k) = (Z_r(k)-rs+sqrt(D(k)))/R(k,k);        % set
      the upper_bound
    while(k~=(m_dimension+1))
        s(k) = search(lower_bound(k),upper_bound(k),
         Constellation_point,s(k));  %check if Sk is in bound
```

```
        if (s(k)==0)         % do not find the point
            k = k+1;         % back to the higher level
        else
            num_nodes = num_nodes+1;
            if (k==1)        % reach the lowest level
                b = norm(Y_r-H_r*s)^2;  % ML detection
                if(b<a)      % Find the smaller node
                    a = b;
                    det_node = s;       % Save the detected node
                end
            else
                k = k-1;     % keep searching the lower level
                RS = 0;
                for j = (k+1):m_dimension
                    RS = RS+R(k+1,j)*s(j);     % Sumation of R*S
                end
                D(k) = D(k+1)-(Z_r(k+1)-RS)^2; % reduce the
                 redius
                break;                         % calculate the
                 searching bound, start the searching again
            end
        end
    end
    if (k==(m_dimension+1)) % the search level is out of the
     node bound
        break;               % terminate the algorithm
    end
end
% Symbol Errors
for i = 1:length(S_r)
    if det_node(i)~=S_r(i)
        error = error+1;    % count the error symbol
    end
end
```

A.7 Schnorr-Euchner (SE) Sphere Detection Algorithm

```
% Before the algorithm, System should be real-valued
S_r = [real(s);imag(s)];    % Generate the real version
H_r = [real(H) -1*imag(H);imag(H) real(H)];
noise_r = [real(noise);imag(noise)];
```

```
Y_r = H_r*S_r+noise_r;      % The real system
[Q,R] = qr(H_r);            % QR decomposition
for k = 1:length(Y_r)
    if (R(k,k)<0)
        Q(:,k) = Q(:,k)*(-1);
         R(k,:) = R(k,:)*(-1);
        end
end
Z_r = Q'*Y_r;

% The radius for SE-SD
d = 2^10;

function [det_node,num_nodes,error] = SE_SD(m_dimension,Q,L,d,
 Y_r,S_r,Constellation_point)

% Input:  m_dimension: search level
%         Q: search level
%         L: the upper triangular from the QR factorization
%         d: the search radius
%         Y_r: real-valued received signal y
%         S_r: real-valued transmitted signal s
%         Constellation_point: real-valued constellation
% Output: det_node: the matrix of the detected node at each
 level
%         num_nodes: the number of the expanded nodes
%         error: number of erroneously detected symbols

for p = 1:(length(Constellation_point)-1)
    partition(p) = (Constellation_point(p)+Constellation_point(p
     +1))/2;
end
gap = Constellation_point(2)-Constellation_point(1);
i = m_dimension;        % search level
bestdist = d;           % the radius matrix
dist(i) = 0;
e(i,:) = Y_r'*Q*L;
[Index,u(i)]=quantiz(e(i,i),partition,Constellation_point);
y_h = (e(i,i)-u(i))/L(i,i);
step(i) = sign(y_h);
num_nodes = 0;            % the counter for the expanded nodes
```

```
s = zeros(m_dimension,1); % initialze the detected result
error = 0;
while (1)
    newdist = dist(i)+y_h^2;
    if (newdist<bestdist)
        num_nodes = num_nodes+1;
        if (i>1)
            for j = 1:i-1
                e(i-1,j) = e(i,j)-y_h*L(i,j);
            end
            i = i-1;
            dist(i) = newdist;
            [Index,u(i)]=quantiz(e(i,i),partition,
             Constellation_point);
            y_h = (e(i,i)-u(i))/L(i,i);
            step(i) = sign(y_h);
        else
            det_node = u;
            bestdist = newdist;
            i = i+1;
            y_h = 2^5;
            for k = 1:2
                u(i) = u(i)+gap*step(i);
                step(i) = (-1)*step(i)-sign(step(i));
                if (isempty(find(u(i)==Constellation_point))==0)
                    y_h = (e(i,i)-u(i))/L(i,i);
                    break;
                end
            end
        end
    else
        if (i==m_dimension)
            return;
        else
            i = i+1;
            y_h = 2^5;
            for k = 1:2
                u(i) = u(i)+gap*step(i);
                step(i) = (-1)*step(i)-sign(step(i));
                if (isempty(find(u(i)==Constellation_point))==0)
                    y_h = (e(i,i)-u(i))/L(i,i);
```

```
                    break;
                end
            end
        end
    end
end
% Symbol Errors
for i = 1:length(S_r)
    if det_node(i)~=S_r(i)
        error = error+1;    % count the error symbol
    end
end
```

A.8 K-Best Sphere Detection Algorithm

```
% Before the algorithm, System should be real-valued
S_r = [real(s);imag(s)];    % Generate the real version
H_r = [real(H) -1*imag(H);imag(H) real(H)];
noise_r = [real(noise);imag(noise)];
Y_r = H_r*S_r+noise_r;      % The real system
[Q,R] = qr(H_r);            % QR decomposition
for k = 1:length(Y_r)
    if (R(k,k)<0)
        Q(:,k) = Q(:,k)*(-1);
         R(k,:) = R(k,:)*(-1);
        end
end
Z_r = Q'*Y_r;

% Choose the value of K

function [det_node,num_nodes,error] = K_SD(m_dimension,R,Z,K,Y_r
 ,H_r,S_r,Constellation_point)

% Input:  m_dimension: search level
%         R: upper triangular matrix with non-negative diagnoal
 element
%         Z: real-valued Z
%         K: the number of the selected best node
%         Y_r: real-valued received signal
%         H_r: real-valued Channel matrix
```

```
%          Constellation_point: real-valued constellation
% Output: det_node: the matrix of the detected node at each
  level
%          num_nodes: the number of the expanded nodes
%          error: number of erroneously detected symbols

num_nodes = 0;
T = [];
s_h = zeros(m_dimension,K);
temp_s = zeros(m_dimension,K);
e = [];
for i = m_dimension:-1:1
    if (i==m_dimension)         % m_dimension-th node
        if (K>length(Constellation_point))
            K1 = length(Constellation_point);
        else
            K1 = K;
        end
        temp_T = zeros(1,K1);
        for j = 1:length(Constellation_point)
            temp_T(j) = (Z(i)-R(i,i)*Constellation_point(j))^2;
                   % Branch cost
        end
        Sort_T = sort(temp_T,'ascend');      % Sort the branch
         cost with the ascend order
        T(i,1:K1) = Sort_T(1:K1);            % Select K partial
         vectors which have the smallest PEDs
        num_nodes = num_nodes+length(T(i,:));
        for t = 1:K1
            s_h(i,t) = Constellation_point(find(temp_T==T(i,t)))
              ;     % save the detected nodes
        end
        temp_s = s_h;
    else                            % i-th node(i<m_dimension)
        count = 1;
        if (K>(length(Constellation_point))^(m_dimension-i))
            K1 = (length(Constellation_point))^(m_dimension-i);
            if (K>(length(Constellation_point))^(m_dimension-i
              +1))
                K2 = length(Constellation_point)^(m_dimension-i
                  +1);
```

```
    else
        K2 = K;
    end
else
    K1 = K;
    K2 = K;
end
length_T = K1*length(Constellation_point);
temp_T = zeros(1,length_T);
for t=1:K1
    for j = 1:length(Constellation_point)    % Go
      through all the constellation nodes
        temp_s(i,t) = Constellation_point(j);
        temp_vector(:,count) = temp_s(:,t);
        rs = 0;
        for n = i:m_dimension
            rs = rs+R(i,n)*temp_s(n,t);      % Calculate
              the branch cost for each level
        end
        e(i,count) = (Z(i)-rs)^2;
        temp_T(count) = T(i+1,t)+e(i,count); % Calculate
          he PED
        count  = count+1;
    end
end
Sort_T = sort(temp_T,'ascend');              % Sort the
 branch cost with the ascend order
T(i,1:K2) = Sort_T(1:K2);                    % Select K
 partial vectors which have the smallest PEDs
num_nodes = num_nodes+length(T(i,:));
for t = 1:K2                                 % Pick the
 nodes retated to the partial vectors
    subscript(t) = find(temp_T==T(i,t));
end
subscript = sort(subscript,'ascend');
for q = 1:K2
    T(i,q) = temp_T(subscript(q));
    s_h(:,q) = temp_vector(:,subscript(q));  % Save the
      detected nodes and Update the path
end
temp_s = s_h;
```

```
    end
end
% Reach the lowest level
for k = 1:K
    b(k) = norm(Y_r-H_r*s_h(:,k))^2;  % Calculate K PEDs
end
det_node = s_h(:,(find(b==min(b))));  % Pick the vector which
 has the smallest PED
% Symbol Errors
for i = 1:length(S_r)
    if det_node(i)~=S_r(i)
        error = error+1;        % count the error symbol
    end
end
```

~

Appendix B

Source Codes for Parallel MIMO Detection Algorithms

B.1 Main Function for Different Detection Algorithms

```
global NumParallel

M_transmit = 4;
M_receive = 4;
m_dimension = 2*M_transmit;  % Channel layer
NumSymbs = 1000;
NumSymbsPerChan = 10;
NumParallel = 1024;    % Degree of Parallelism

% Define the Constellation Point
M_QAM = 16;
Energy = 2*(M_QAM-1)/3;
SNR = 5:5:40;
LengthSNR = length(SNR);
MaxErrorLimit = 500;
MinErrorLimit = 100;

bbb = gsingle([-3.0000 + 3.0000i;
               -3.0000 + 1.0000i;
               -3.0000 - 3.0000i;
               -3.0000 - 1.0000i;
               -1.0000 + 3.0000i;
               -1.0000 + 1.0000i;
```

```
               -1.0000 - 3.0000i;
               -1.0000 - 1.0000i;
               3.0000 + 3.0000i;
               3.0000 + 1.0000i;
               3.0000 - 3.0000i;
               3.0000 - 1.0000i;
               1.0000 + 3.0000i;
               1.0000 + 1.0000i;
               1.0000 - 3.0000i;
               1.0000 - 1.0000i]);

constellation_point = [-3 -1 1 3];
Constellation_point = gsingle(constellation_point);
partition = gzeros(1,length(Constellation_point)-1);
for p = 1:(length(Constellation_point)-1)
    partition(p) = (Constellation_point(p)+Constellation_point(p
     +1))/2;
end

for ee = 1:LengthSNR
    snr = 10^(SNR(ee)/10);
    NoiseScale = sqrt(Energy*M_transmit/(2*snr));
    ChanAge = NumSymbsPerChan;   % force generation of first
     channel matrix

    Error_temp = zeros(NumSymbs,1);

    for ss = 1:NumSymbs
        x = gsingle(randi([0,M_QAM-1],M_transmit,1,NumParallel))
         ;
        s = bbb(x+1);

        ChanAge = ChanAge + 1;
        if (ChanAge > NumSymbsPerChan)
            % It is time to generate a new channel matrix
            ChanAge = 0;
            H = complex(grandn(M_receive,M_transmit,NumParallel)
             ,grandn(M_receive,M_transmit,NumParallel))/sqrt(2)
             ;
        end   % if (ChanAge > NumSymbsPerChan)
```

```
        n = complex(grandn(M_receive,1,NumParallel),grandn(
          M_receive,1,NumParallel))*sqrt(Energy*M_transmit/(2*
          snr));

        gfor pp = 1:NumParallel
            y(:,:,pp) = H(:,:,pp)*s(:,:,pp) + n(:,:,pp);
        gend

        % Detection algorithm

        % Calculate the symbol error rate
        Error_temp(ss) = Error_temp(ss)+error/(2*M_transmit*
          NumParallel);
    end
    SER(i) = sum(Error_temp)/NumSymbs;
end
```

B.2 New Matrix Inverse Function

```
function  result = NewInverse(A)

% Input:       A: N x N x NumParallel matrix, it is a GPU
  structure
% Output: result: N x N x NumParallel inverse matrix of A, it is
   a GPU structure
% Initialize X, Y, the temporary storage matrix C, and the row
% permutation information matrix R

global NumParallel

IdentityMat = geye(M_transmit);
[~,N] = size(A(:,:,1));
B = IdentityMat;   %B is an N x N identity matrix
X = gzeros(N,N,NumParallel);
Y = gzeros(N,N,NumParallel);
R = gsingle(1:N);
R = repmat(R,NumParallel,1);

C = gzeros(1,N,NumParallel);
j = gzeros(NumParallel,1);
d = gzeros(NumParallel,1);
```

```
mult = gzeros(NumParallel,1);
% The next steps is to find the factorization (LU decomposition)
gfor pp = 1:NumParallel
        B(:,:,pp) = IdentityMat;
        for p = 1:N-1
        %Find the pivot row for column p
        [~, j(pp)] = max(abs(A(p:N,p,pp)));

        %Interchange row p and j
        C(:,:,pp) = A(p,:,pp);
        A(p,:,pp) = A(j(pp)+p-1,:,pp);
        A(j(pp)+p-1,:,pp) = C(:,:,pp);
        d(pp) = R(pp,p);
        R(pp,p) = R(pp,j(pp)+p-1);
        R(pp,j(pp)+p-1) = d(pp);

        %Calculate multiplier and place in subdiagonal portion
         of A
        for k = p+1:N
                mult(pp) = A(k,p,pp)/A(p,p,pp);
                A(k,p,pp) = mult(pp);
                A(k,p+1:N,pp) = A(k,p+1:N,pp)-mult(pp)*A(p,p+1:N
                  ,pp);
        end
        end

        % Solve the linear equation (LUX=B)
        for q = 1:N
        %Solve LY = B for Y(:,q)
        Y(1,q,pp) = B(R(pp,1),q,pp);
                for k = 2:N
                Y(k,q,pp) = B(R(pp,k),q,pp)-A(k,1:k-1,pp)*Y(1:k
                  -1,q,pp);
        end
        %Solve UX = Y for X(:,q)
        X(N,q,pp) = Y(N,q,pp)/A(N,N,pp);
        for k = N-1:-1:1
                X(k,q,pp) = (Y(k,q,pp)-A(k,k+1:N,pp)*X(k+1:N,q,
                 pp))/A(k,k,pp);
        end
```

```
        end
gend
result = X;
```

B.3 Zero Forcing (ZF) Detection Algorithm Parallel Version

```
function SymbolError = ZFParallel(y,s,Constellation_point,
 partition)

% Input:  y: complex-valued received signal
%         s: complex-valued transmitted signal
%         Constellation_point: real-valued constellation
%         partition: constellation points partition
% Output: SymbolError: number of erroneously detected symbols

global NumParallel
RealCount = gzeros(NumParallel,1);
Realcondition = gzeros(NumParallel,1);
ImgCount = gzeros(NumParallel,1);
Imgcondition = gzeros(NumParallel,1);
error = gzeros(1,NumParallel);

gfor pp = 1:NumParallel
        transpose_h(:,:,pp) = h(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*h(:,:,pp);
gend
AfterInverseMat = NewInverse(ZF_InverseMat);
gfor pp = 1:NumParallel
        Q(:,:,pp) = AfterInverseMat(:,:,pp)*transpose_h(:,:,pp);
gend

gfor pp = 1:NumParallel
% Nulling
TempVec(:,:,pp) = Q(:,:,pp)*Y(:,:,pp);
% Slicing
[~,R(:,:,pp)] = quantization(real(TempVec(:,:,pp)),partition,
 Constellation_point);
[~,Img(:,:,pp)] = quantization(imag(TempVec(:,:,pp)),partition,
 Constellation_point);
```

```
gend
% Symbol Errors
for i = 1:length(s(:,1,1))
    gfor pp = 1:NumParallel
    RealCount(pp) = R(:,i,pp)-real(s(i,:,pp));
    Realcondition(pp) = RealCount(pp)~=0;
    error(pp) = error(pp)+Realcondition(pp);
    ImgCount(pp) = Img(:,i,pp)-imag(s(i,:,pp));
    Imgcondition(pp) = ImgCount(pp)~=0;
    error(pp) = error(pp)+Imgcondition(pp);
    gend
end

SymbolError = single(sum(error));     % Cast GPU data back to
 CPU
```

B.4 Minimum Mean Square Error (MMSE) Detection Algorithm Parallel Version

```
function SymbolError = MMSEParallel(Y,s,Constellation_point,
 partition)

% Input:  y: complex-valued received signal
%         s: complex-valued transmitted signal
%         Constellation_point: real-valued constellation
%         partition: constellation points partition
% Output: SymbolError: number of erroneously detected symbols

global NumParallel
RealCount = gzeros(NumParallel,1);
Realcondition = gzeros(NumParallel,1);
ImgCount = gzeros(NumParallel,1);
Imgcondition = gzeros(NumParallel,1);
error = gzeros(1,NumParallel);

IdentityMat = geye(M_transmit);

gfor pp = 1:NumParallel
        transpose_h(:,:,pp) = h(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*h(:,:,pp)+(1/
```

```
        snr)*IdentityMat;
gend
AfterInverseMat = NewInverse(InverseMat);
gfor pp = 1:NumParallel
        Q(:,:,pp) = AfterInverseMat(:,:,pp)*transpose_h(:,:,pp);
gend

gfor pp = 1:NumParallel
% Nulling
TempVec(:,:,pp) = Q(:,:,pp)*Y(:,:,pp);
% Slicing
[~,R(:,:,pp)] = quantization(real(TempVec(:,:,pp)),partition,
 Constellation_point);
[~,Img(:,:,pp)] = quantization(imag(TempVec(:,:,pp)),partition,
 Constellation_point);
gend
% Symbol Errors
for i = 1:length(s(:,1,1))
    gfor pp = 1:NumParallel
    RealCount(pp) = R(:,i,pp)-real(s(i,:,pp));
    Realcondition(pp) = RealCount(pp)~=0;
    error(pp) = error(pp)+Realcondition(pp);
    ImgCount(pp) = Img(:,i,pp)-imag(s(i,:,pp));
    Imgcondition(pp) = ImgCount(pp)~=0;
    error(pp) = error(pp)+Imgcondition(pp);
    gend
end

SymbolError = single(sum(error));      % Cast GPU data back to
 CPU
```

B.5 V-BLAST Detection Algorithm Parallel Version

```
function SymbolError = VBLASTParallel(snr,M_transmit,H,y,s,
 Constellation_point,partition)

% Input:  snr: signal-to-noise ratio
%         M_transmit: number of transmit antennas
%         H: complex-valued channel matrix
%         y: complex-valued received signal
%         s: complex-valued transmitted signal
```

```
%         Constellation_point: real-valued constellation
%         partition: constellation points partition
% Output: SymbolError: number of erroneously detected symbols

global NumParallel

M_receive = length(H(:,1,1));
k = gzeros(1,M_transmit,NumParallel);
TestY = Y;
TestH = H;
error = gzeros(1,NumParallel);
shk = gzeros(1,NumParallel);
R = gzeros(NumParallel,1);
RealCount = gzeros(NumParallel,1);
Realcondition = gzeros(NumParallel,1);
Img = gzeros(NumParallel,1);
ImgCount = gzeros(NumParallel,1);
Imgcondition = gzeros(NumParallel,1);

transpose_h = gzeros(M_transmit,M_receive,NumParallel,'single');
InverseMat = gzeros(M_transmit,M_transmit,NumParallel,'single');
I = gzeros(NumParallel,1);
NormQ = gzeros(1,M_transmit,NumParallel);

IdentityMat = geye(M_transmit);

gfor pp = 1:NumParallel
        transpose_h(:,:,pp) = h(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*h(:,:,pp)+(1/
          snr)*IdentityMat;
gend
AfterInverseMat = NewInverse(InverseMat);
gfor pp = 1:NumParallel
        G(:,:,pp) = AfterInverseMat(:,:,pp)*transpose_h(:,:,pp);
gend

for i = 1:M_transmit
        gfor pp = 1:NumParallel
                for p = 1:M_transmit
                        NormQ(1,p,pp) = sum(abs(G(p,:,pp)).^2);
                end
```

```
        [~,I(pp)] = min(NormQ(:,:,pp));   % I is the subscript
         of the minimum value
        k(1,i,pp) = I(pp);       % save the subscript

                % Nulling
        shk(pp) = G(I(pp),:,pp)*TestY(:,:,pp);
        % Slicing
        [~,R(pp)] = quantization(real(shk(pp)),partition,
         Constellation_point);
        RealCount(pp) = R(pp)-real(s(I(pp),:,pp));
        Realcondition(pp) = RealCount(pp)~=0;
        error(pp) = error(pp)+Realcondition(pp);

        [~,Img(pp)] = quantization(imag(shk(pp)),partition,
         Constellation_point);
        ImgCount(pp) = Img(pp)-imag(s(I(pp),:,pp));
        Imgcondition(pp) = ImgCount(pp)~=0;
        error(pp) = error(pp)+Imgcondition(pp);

        TestY(:,:,pp) = TestY(:,:,pp)-(R(pp)+1i*Img(pp))*TestH
         (:,I(pp),pp);  % interference cancellation
        TestH(:,I(pp),pp) = 0;    % set the used channel into 0

        transpose_h(:,:,pp) = TestH(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*TestH(:,:,pp)
         +(1/snr)*geye(M_transmit);
        gend
        AfterInverseMat = NewInverse(InverseMat);
        gfor pp = 1:NumParallel
                G(:,:,pp) = AfterInverseMat(:,:,pp)*transpose_h
                  (:,:,pp);
        gend
end

SymbolError = single(sum(error)); % Cast GPU data back to CPU
```

B.6 Parallel V-BLAST Detection Algorithm

```
function SymbolError = Parallel_VBLAST(snr,M_transmit,M_receive,
 H,y,s,Constellation_point,ConsMat_s,M_QAM,partition)
```

```
% Input:  snr: signal-to-noise ratio
%         M_transmit: number of transmit antennas
%         M_receive: number of received antennas
%         H: complex-valued channel matrix
%         y: complex-valued received signal
%         s: complex-valued transmitted signal
%         Constellation_point: real-valued constellation
%         ConsMat_s: complex-valued constellation
%         M_QAM : size of the complex-valued constellation
%         partition: constellation points partition
% Output: SymbolError: number of erroneously detected symbols

global NumParallel
error = gzeros(1,NumParallel);

HH = H;
tempH = H;
YY = Y;
s_det = gzeros(M_transmit,NumParallel);
size = length(s(:,1,1));
G = gzeros(M_transmit,M_receive,M_transmit,NumParallel);
shk = gzeros(1,NumParallel);
SymbolTest = gzeros(M_transmit,M_QAM,NumParallel);
TempY = gzeros(M_transmit,M_QAM,NumParallel);
TempError = gzeros(1,M_QAM,NumParallel);
NormQ = gzeros(1,M_transmit,NumParallel);
VBLASTI = gzeros(1,NumParallel);
R = gzeros(1,NumParallel);
Img = gzeros(1,NumParallel);
order = gzeros(1,M_transmit,NumParallel);

RealCount = gzeros(1,NumParallel);
Realcondition = gzeros(1,NumParallel);
ImgCount = gzeros(1,NumParallel);
Imgcondition = gzeros(1,NumParallel);
ConsMat = repmat(ConsMat_s,[1,NumParallel]);

IdentityMat = geye(M_transmit);

gfor pp = 1:NumParallel
```

```
        transpose_h(:,:,pp) = h(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*h(:,:,pp)+(1/
          snr)*IdentityMat;
gend
AfterInverseMat = NewInverse(InverseMat);
gfor pp = 1:NumParallel
        Q(:,:,pp) = AfterInverseMat(:,:,pp)*transpose_h(:,:,pp);
gend

gfor pp = 1:NumParallel
    G(:,:,1,pp) = Q(:,:,pp);
    for p = 1:M_transmit
        NormQ(1,p,pp) = (norm(G(p,:,1,pp)))^2;  % calculate the
          normal value of G
    end
    [~,I(pp)] = max(NormQ(:,:,pp));    % I is the subscript of
      the maximum value
    tempH(:,I(pp),pp) = 0;

    transpose_h(:,:,pp) = tempH(:,:,pp)';
    InverseMat(:,:,pp) = transpose_h(:,:,pp)*tempH(:,:,pp)+(1/
      snr)*geye(M_transmit);
gend
AfterInverseMat = NewInverse(InverseMat);
gfor pp = 1:NumParallel
        G(:,:,1,pp) = AfterInverseMat(:,:,pp)*transpose_h(:,:,pp
          );
gend

VBLASTk = gzeros(1,M_transmit-1,NumParallel);

% Calculate the Norm values for each channel layer
for jj = 1:M_transmit-1
        gfor pp = 1:NumParallel
        NormQ(1,I(pp),pp) = Inf;
        for t = 1:jj-1
            NormQ(1,VBLASTk(1,t,pp),pp) = Inf;   % set the
              detected normal value to infinity
        end
        [~,VBLASTI(pp)] = min(NormQ(:,:,pp));
        VBLASTk(1,jj,pp) = VBLASTI(pp);
```

```
        tempH(:,VBLASTI(pp),pp) = 0;
        transpose_h(:,:,pp) = tempH(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*tempH(:,:,pp)
         +(1/snr)*geye(M_transmit);
    gend
    AfterInverseMat = NewInverse(InverseMat);
    gfor pp = 1:NumParallel
            G(:,:,jj+1,pp) = AfterInverseMat(:,:,pp)*
             transpose_h(:,:,pp);
            for qq = 1:M_receive
                    NormQ(1,qq,pp) = (norm(G(qq,:,jj+1,pp)))
                     ^2;   % calculate the normal value of
                     G
            end
        gend
end

% Fully Enumerate the weakest layer within M_QAM
for tt = 1:M_QAM
        TestY = Y;
        TestH = H ;
        gfor pp = 1:NumParallel
        TestY(:,:,pp) = TestY(:,:,pp)-ConsMat(tt,pp)*TestH(:,I(
         pp),pp);
        TestH(:,I(pp),pp) = 0;
        gend
        SymbolTestRow = gzeros(NumParallel,M_transmit);
        VBLASTSymbolTest = gzeros(M_transmit-1,NumParallel);

        gfor pp = 1:NumParallel
                for jj = 1:M_transmit-1
                        % nulling
                        shk(:,pp) = G(VBLASTk(1,jj,pp),:,jj,pp)*
                         TestY(:,:,pp);
                        % Slicing
                        [~,R(:,pp)] = quantization(real(shk(:,pp
                         )),partition,Constellation_point);
                        [~,Img(:,pp)] = quantization(imag(shk(:,
                         pp)),partition,Constellation_point);

                        VBLASTSymbolTest(jj,pp) = R(:,pp)+1i*Img
```

```
                    (:,pp);         % get the real detected
                     symbol

                  % interference cancellation
                  TestY(:,:,pp) = TestY(:,:,pp)-(R(:,pp)+1
                    i*Img(:,pp))*TestH(:,VBLASTk(1,jj,pp),
                    pp);
                  TestH(:,VBLASTk(1,jj,pp),pp) = 0;
                            % set the used channel into 0
            end
            SymbolTestRow(pp,:) = [ConsMat(tt,pp)
             VBLASTSymbolTest(:,pp).'];
            order(:,:,pp) = [I(pp) VBLASTk(:,:,pp)];
        gend

        TempVector = ColumnExchange(SymbolTestRow,order);

        gfor pp = 1:NumParallel
        SymbolTest(:,tt,pp) = (TempVector(pp,:)).';
        TempY(:,tt,pp) = HH(:,:,pp)*SymbolTest(:,tt,pp);
        TempError(1,tt,pp) = norm(YY(:,:,pp)-TempY(:,tt,pp))^2;
        gend
end

gfor pp = 1:NumParallel
        [~,Index(pp)] = min(TempError(:,:,pp));
        s_det(:,pp) = SymbolTest(:,Index(pp),pp);

        % calculate the SER
        for i = 1:size
                RealCount(pp) = real(s_det(i,pp))-real(s(i,1,pp)
                 );
                Realcondition(pp) = RealCount(pp)~=0;
                error(pp) = error(pp)+Realcondition(pp);
                ImgCount(pp) = imag(s_det(i,pp))-imag(s(i,1,pp))
                 ;
                Imgcondition(pp) = ImgCount(pp)~=0;
                error(pp) = error(pp)+Imgcondition(pp);
        end
gend
```

```
SymbolError = single(sum(error));      % Cast GPU data back to
 CPU
```

B.7 K-Best Sphere Detection Algorithm Parallel Version

```
% Before the algorithm, System should be real-valued
S_r = [real(s);imag(s)];      % Generate the real version
H_r = [real(H) -1*imag(H);imag(H) real(H)];
noise_r = [real(noise);imag(noise)];

gfor pp = 1:NumParallel
        [Q(:,:,pp),R(:,:,pp)] = qr(H_r(:,:,pp));              %
         QR factorization
        for k = 1:2*M_receive
                QRCondition(pp) = R(k,k,pp)<0;
                Q(:,k,pp) = (1-QRCondition(pp))*Q(:,k,pp)+
                 QRCondition(pp)*Q(:,k,pp)*(-1);
                R(k,:,pp) = (1-QRCondition(pp))*R(k,:,pp)+
                 QRCondition(pp)*R(k,:,pp)*(-1);
        end
        Y_r(:,:,pp) = H_r(:,:,pp)*S_r(:,:,pp)+noise_r(:,:,pp); %
          The real system
        Z_r(:,:,pp) = Q(:,:,pp)'*Y_r(:,:,pp);
gend

function SymbolError = K_SDParallel(m_dimension,R,Z,K,Y_r,H_r,
 S_r,Constellation_point)

% Input:  m_dimension: search level
%         R: upper triangular matrix with non-negative diagnoal
 element
%         Z: real-valued Z
%         K: the number of the selected best node
%         Y_r: real-valued received signal
%         H_r: real-valued Channel matrix
%         Constellation_point: real-valued constellation
% Output: SymbolError: number of erroneously detected symbols

global NumParallel
```

```
LengthConstelaltion = length(Constellation_point(1,:));
LengthKConstelaltion = K*LengthConstelaltion;
T = gzeros(m_dimension,K,NumParallel);
s_h = gzeros(m_dimension,K,NumParallel);
e = gzeros(m_dimension,LengthKConstelaltion,NumParallel);
temp_vector = gzeros(m_dimension,LengthKConstelaltion,
 NumParallel);
subscript = gzeros(1,K,NumParallel);

b = gzeros(1,K,NumParallel);

error = gzeros(NumParallel,1);

i = m_dimension;
KCondition_0 = K>LengthConstelaltion;
K1 = KCondition_0*LengthConstelaltion+(1-KCondition_0)*K;
temp_T = gzeros(1,LengthConstelaltion,NumParallel);
gfor pp = 1:NumParallel
        for j = 1:LengthConstelaltion
        temp_T(1,j,pp) = (Z(i,:,pp)-R(i,i,pp)*
         Constellation_point(pp,j))^2;       % Branch cost
        end
        Sort_T = sort(temp_T,'ascend');          % Sort the
         branch cost with the ascend order
        T(i,1:K1,pp) = Sort_T(1,1:K1,pp);       % Select K
         partial vectors which have the smallest PEDs
        for t = 1:K1
        s_h(i,t,pp) = Constellation_point(pp,FindData(T(i,t,pp),
         temp_T(:,:,pp)));  % save the detected nodes
        end
        emp_s = s_h;

        for i = m_dimension-1:-1:1
        count = 1;
        KCondition = K>(LengthConstelaltion)^(m_dimension-i);
        K1 = KCondition*(LengthConstelaltion)^(m_dimension-i)
         +(1-KCondition)*K;
        KCondition_1 = K>(LengthConstelaltion)^(m_dimension-i+1)
         ;
        K2 = KCondition*(KCondition_1*LengthConstelaltion^(
```

```
 m_dimension-i+1)+(1-KCondition_1)*K)+(1-KCondition)*K;

temp_T = gzeros(1,K1*LengthConstelaltion,NumParallel);
for t=1:K1
    for j = 1:LengthConstelaltion   % Go through all the
      constellation nodes
        temp_s(i,t,pp) = Constellation_point(pp,j);
        temp_vector(:,count,pp) = temp_s(:,t,pp);

        e(i,count,pp) = (Z(i,1,pp)-R(i,i:m_dimension,pp)
         *temp_s(i:m_dimension,t,pp))^2;
        temp_T(1,count,pp) = T(i+1,t,pp)+e(i,count,pp);
           % Calculate he PED
        count  = count+1;
    end
end
Sort_T = sort(temp_T,'ascend');     % Sort the branch
 cost with the ascend order
T(i,1:K2,pp) = Sort_T(1,1:K2,pp);   % Select K partial
 vectors which have the smallest PEDs
for t = 1:K2                        % Pick the nodes
 retated to the partial vectors
    subscript(1,t,pp) = FindData(T(i,t,pp),temp_T(:,:,pp
     ));
end
subscript = sort(subscript,'ascend');
T(i,1:K2,pp) = temp_T(:,subscript(1,1:K2,pp),pp);
s_h(:,1:K2,pp) = temp_vector(:,subscript(1,1:K2,pp),pp);
   % Save the detected nodes and Update the path
temp_s = s_h;
end
% Reach the lowest level
for k = 1:K
b(1,k,pp) = norm(Y_r(:,:,pp)-H_r(:,:,pp)*s_h(:,k,pp))^2;
   % Calculate K PEDs
end
det_node(:,:,pp) = s_h(:,FindMinimum(b(:,:,pp)),pp);
 % Pick the vector has the smallest PED and save it

error(pp) = sum(det_node(1:m_dimension,:,pp)~=S_r(1:
 m_dimension,:,pp));
```

```
gend

SymbolError = single(sum(error));     % Cast GPU data back to
 CPU
```

B.8 Parallel V-BLAST Detection Algorithm with Real and Imaginary Components

```
% Before the algorithm, System should be real-valued
S_r = [real(s);imag(s)];      % Generate the real version
H_r = [real(H) -1*imag(H);imag(H) real(H)];
noise_r = [real(noise);imag(noise)];

IdentityMat = geye(2*M_transmit);

gfor pp = 1:NumParallel
        transpose_h(:,:,pp) = H_r(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*H_r(:,:,pp)+(1/
         snr)*IdentityMat;
gend
AfterInverseMat = NewInverse(InverseMat);
gfor pp = 1:NumParallel
        Q(:,:,pp) = AfterInverseMat(:,:,pp)*transpose_h(:,:,pp);
        Y_r(:,pp) = H_r(:,:,pp)*S_r(:,pp)+noise_r(:,pp);      %
         The real system
gend

function SymbolError = Parallel_VBLAST_Real(snr,M_transmit,H_r,
 Y_r,S_r,Constellation_point,partition)

% Input:  snr: signal-to-noise ratio
%         M_transmit: number of transmit antennas
%         H_r: real-valued channel matrix
%         Y_r: real-valued received signal y
%         S_r: real-valued transmitted signal s
%         Constellation_point: real-valued constellation
%         partition: constellation points partition
% Output: SymbolError: number of erroneously detected symbols

global pp
```

```
global NumParallel
gfor pp = 1:NumParallel
    HH(:,:,pp) = H;
    RealTempH(:,:,pp) = H;
    ImagTempH(:,:,pp) = H;
    YY(:,pp) = Y;
    M_receive = (length(Y(:,1)))/2;
    RealG = gzeros(2*M_transmit,2*M_receive,2*M_transmit,
     NumParallel);
    ImagG = gzeros(2*M_transmit,2*M_receive,2*M_transmit,
     NumParallel);

    Dimention = length(Constellation_point);
    TempY = gzeros(2*M_receive,2*Dimention,NumParallel);     %
     all the suspected Y
    TempError = gzeros(1,2*Dimention);    % error between
     original Y and all the suspected Y
    NormQ = gzeros(1,M_transmit);
    RealNormQ = gzeros(1,2*M_transmit);
    ImagNormQ = gzeros(1,2*M_transmit);

        G(:,:,pp) = Q;

    for p = 1:M_transmit
        NormQ(p) = (norm(G(p,:,pp)))^2;   % calculate the normal
          value of G
    end

    [~,I_Real] = max(NormQ);              % Weakest Channel
     Layer
    RealTempH(:,I_Real,pp) = 0;

    transpose_h(:,:,pp) = RealTempH(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*RealTempH(:,:,
         pp)+(1/snr)*IdentityMat;
gend
AfterInverseMat = NewInverse(InverseMat);
gfor pp = 1:NumParallel
        RealG(:,:,1,pp) = AfterInverseMat(:,:,pp)*transpose_h
         (:,:,pp);
    for qq = 1:2*M_transmit
```

```
        RealNormQ(qq) = (norm(RealG(qq,:,1,pp)))^2;        %
         calculate the normal value of G
    end

    I_Imag = I_Real+Dimention;
    ImagTempH(:,I_Imag,pp) = 0;

        transpose_h(:,:,pp) = ImagTempH(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*ImagTempH(:,:,
         pp)+(1/snr)*IdentityMat;
gend
AfterInverseMat = NewInverse(InverseMat);
gfor pp = 1:NumParallel
        ImagG(:,:,1,pp) = AfterInverseMat(:,:,pp)*transpose_h
         (:,:,pp);
    for qq = 1:2*M_transmit
        ImagNormQ(qq) = (norm(ImagG(qq,:,1,pp)))^2;        %
         calculate the normal value of G
    end
gend
%%%%%%%%%%%%%%%%%
% Real part Norm of G
RealVBLASTk = gzeros(1,2*M_transmit-1);
for jj = 1:2*M_transmit-1
        RealNormQ(I_Real) = Inf;
        for t = 1:jj-1
                RealNormQ(RealVBLASTk(t)) = Inf; % set the
                 detected normal value to infinity
        end
        [~,Real] = min(RealNormQ);           % I is the
         subscript of the minimum value
        RealVBLASTk(jj) = Real;
        gfor pp = 1:NumParallel
                RealTempH(:,RealVBLASTk(jj),pp) = 0;

                transpose_h(:,:,pp) = RealTempH(:,:,pp)';
                InverseMat(:,:,pp) = transpose_h(:,:,pp)*
                 RealTempH(:,:,pp)+(1/snr)*IdentityMat;
        gend
        AfterInverseMat = NewInverse(InverseMat);
        gfor pp = 1:NumParallel
```

```
            RealG(:,:,jj+1,pp) = AfterInverseMat(:,:,pp)*
             transpose_h(:,:,pp);

        for qq = 1:length(RealTempH(:,1))
            RealNormQ(qq) = (norm(RealG(qq,:,jj+1,pp)))^2;  %
             calculate the normal value of G
        end
    gend
end
%%%%%%%%%%%%%%%%%
% Imaginary part Norm of G
ImagVBLASTk = gzeros(1,2*M_transmit-1);
for jj = 1:2*M_transmit-1
        ImagNormQ(I_Imag) = Inf;
        for t = 1:jj-1
                ImagNormQ(ImagVBLASTk(t)) = Inf;     % set the
                 detected normal value to infinity
        end
        [~,Imag] = min(ImagNormQ);          % I stands for the
         subscript of the minimum value in the normal value set
        ImagVBLASTk(jj) = Imag;
        gfor pp = 1:NumParallel
                ImagTempH(:,ImagVBLASTk(jj),pp) = 0;

                transpose_h(:,:,pp) = ImagTempH(:,:,pp)';
                InverseMat(:,:,pp) = transpose_h(:,:,pp)*
                 ImagTempH(:,:,pp)+(1/snr)*IdentityMat;
        gend
        AfterInverseMat = NewInverse(InverseMat);
        gfor pp = 1:NumParallel
                ImagG(:,:,jj+1,pp) = AfterInverseMat(:,:,pp)*
                 transpose_h(:,:,pp);

                for qq = 1:length(ImagTempH(:,1))
                        ImagNormQ(qq) = (norm(ImagG(qq,:,jj+1,pp
                         )))^2;        % calculate the normal
                         value of G
                end
        gend
end
gfor pp = 1:NumParallel
```

```
%%%%%%%%%%%%%%%%%%
% Real part V-BLAST
    RealSymbolTestRow = gzeros(Dimention,2*M_transmit,
     NumParallel);
    RealVBLASTSymbolTest = gzeros(2*M_transmit-1,1,NumParallel);
    for tt = 1:Dimention
        TestY = gzeros(2*M_receive,NumParallel);
        TestH = gzeros(2*M_receive,2*M_transmit,NumParallel);
        TestY(:,pp) = Y;
        TestH(:,:,pp) = H;
        TestY(:,pp) = TestY(:,pp)-Constellation_point(tt)*TestH
         (:,I_Real,pp);
        TestH(:,I_Real,pp) = 0;
        for jj = 1:2*M_transmit-1
            shk = gzeros(1,NumParallel);
            % nulling
            shk(:,pp) = RealG(RealVBLASTk(jj),:,jj,pp)*TestY(:,
             pp);
                % Slicing
            [~,RealValue] = quantiz(shk(:,pp),partition,
             Constellation_point);
            RealVBLASTSymbolTest(jj,1,pp) = RealValue;        %
             get the real detected symbol
                % interference cancellation
            TestY(:,pp) = TestY(:,pp)-RealValue*TestH(:,
             RealVBLASTk(jj),pp);
            TestH(:,RealVBLASTk(jj),pp) = 0;    % set the used
             channel into 0
        end
        RealSymbolTestRow(tt,:,pp) = [Constellation_point(tt)
         RealVBLASTSymbolTest(:,1,pp).'];
    end
%%%%%%%%%%%%%%%%%%
% Imaginary part V-BLAST
    ImagSymbolTestRow = gzeros(Dimention,2*M_transmit,
     NumParallel);
    ImagVBLASTSymbolTest = gzeros(2*M_transmit-1,NumParallel);
    for tt = 1:Dimention
        TestY = gzeros(2*M_receive,NumParallel);
        TestH = gzeros(2*M_receive,2*M_transmit,NumParallel);
        TestY(:,pp) = Y;
```

```
        TestH(:,:,pp) = H;
        TestY(:,pp) = TestY(:,pp)-Constellation_point(tt)*TestH
         (:,I_Imag,pp);
        TestH(:,I_Imag,pp) = 0;
        for jj = 1:2*M_transmit-1
            shk = gzeros(1,NumParallel);
            % nulling
            shk(:,pp) = ImagG(ImagVBLASTk(jj),:,jj,pp)*TestY(:,
             pp);
                % Slicing
            [~,ImagValue] = quantiz(shk(:,pp),partition,
             Constellation_point);
            ImagVBLASTSymbolTest(jj,pp) = ImagValue;        %
             get the real detected symbol

            TestY(:,pp) = TestY(:,pp)-ImagValue*TestH(:,
             ImagVBLASTk(jj),pp);      % interference
             cancellation
            TestH(:,ImagVBLASTk(jj),pp) = 0;      % set the used
              channel into 0
        end
        ImagSymbolTestRow(tt,:,pp) = [Constellation_point(tt)
         ImagVBLASTSymbolTest(:,pp).'];
    end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Symbol Errors
    Realorder = [I_Real RealVBLASTk];
    Imagorder = [I_Imag ImagVBLASTk];
    RealTotalSymbol = Symbol_ColumnExchangeBack(
     RealSymbolTestRow(:,:,pp),Realorder);
    ImagTotalSymbol = Symbol_ColumnExchangeBack(
     ImagSymbolTestRow(:,:,pp),Imagorder);
    TotalSymbol(:,:,pp) = [RealTotalSymbol(:,:,pp)'
     ImagTotalSymbol(:,:,pp)'];
    for kk = 1:2*Dimention
        TempY(:,kk,pp) = HH(:,:,pp)*TotalSymbol(:,kk,pp);
        TempError(kk) = norm(YY(:,pp)-TempY(:,kk,pp))^2;
    end
    [~,Index] = min(TempError);
    s_det = gzeros(2*M_transmit,NumParallel);
    s_det(:,pp) = TotalSymbol(:,Index,pp);
```

```
    ss(:,pp) = s;
    for i = 1:2*M_transmit
        Count = s_det(i,pp)-ss(i,pp);
        condition = Count~=0;
        error = error+condition;
    end
gend

SymbolError = single(sum(error));     % Cast GPU data back to
 CPU
```

B.9 Fully Enumerated K-Best Detection Algorithm

```
% Before the algorithm, System should be real-valued
S_r = [real(s);imag(s)];      % Generate the real version
H_r = [real(H) -1*imag(H);imag(H) real(H)];
noise_r = [real(noise);imag(noise)];

% Choose the value of K

function SymbolError = Fully_KBest(M_transmit,M_receive,
 m_dimension,K,H_r,Y_ori,snr,Constellation_point)

% Input:  M_transmit: number of transmit antennas
%         M_receive: number of received antennas
%         m_dimension: channel level
%         K: the number of the selected best node
%         H_r: real-valued Channel H
%         Y_ori: real-valued received signal y
%         snr: signal-to-noise ratio
%         Constellation_point: real-valued constellation
% Output: SymbolError: number of erroneously detected symbols

global NumParallel

error = gzeros(NumParallel,1);
num_nodes = 0;
ConstellationSzie = length(Constellation_point);
HH(:,:,pp) = H_r;
RealZ = gzeros(m_dimension,NumParallel);
```

```
ImagZ = gzeros(m_dimension,NumParallel);

IdentityMat = geye(2*M_transmit);

gfor pp = 1:NumParallel
        transpose_h(:,:,pp) = HH(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*HH(:,:,pp)+(1/
         snr)*IdentityMat;
gend
AfterInverseMat = NewInverse(InverseMat);
gfor pp = 1:NumParallel
        Vblast_Q(:,:,pp) = AfterInverseMat(:,:,pp)*transpose_h
         (:,:,pp);
gend

G = Vblast_Q;
NormQ = gzeros(1,M_transmit);
RealOriginal_order = gdouble(1:m_dimension);
ImagOriginal_order = gdouble(1:m_dimension);

gfor pp = 1:NumParallel
    for p = 1:M_transmit
        NormQ(p) = (norm(G(p,:,pp)))^2; % calculate the normal
         value of G
    end
    [~,I_Real] = max(NormQ);            % I is the subscript of
     the maximum value
    I_Imag = I_Real+ConstellationSzie;

    RealTempOrder = RealOriginal_order(m_dimension);
    RealOriginal_order(m_dimension) = I_Real;
    RealOriginal_order(I_Real) = RealTempOrder;

    ImagTempOrder = ImagOriginal_order(m_dimension);
    ImagOriginal_order(m_dimension) = I_Imag;
    ImagOriginal_order(I_Imag) = ImagTempOrder;

    Realnew_H = Channel_ColumnExchange(HH(:,:,pp),
     RealOriginal_order);
    Realnew_H(:,m_dimension,pp) = 0;
    [RealQ(:,:,pp),RealR(:,:,pp)] = qr(Realnew_H(:,:,pp));  % QR
```

```
    factorization
  for k = 1:M_receive
      QRCondition = RealR(k,k,pp)<0;
      RealQ(:,k,pp) = (1-QRCondition)*RealQ(:,k,pp)+
       QRCondition*RealQ(:,k,pp)*(-1);
      RealR(k,:,pp) = (1-QRCondition)*RealR(k,:,pp)+
       QRCondition*RealR(k,:,pp)*(-1);
  end
  Y_r = Y_ori;
%%%%%%%%%%%%%%%%%%%%
  RealT = gzeros(m_dimension,K);
  Reals_h = gzeros(m_dimension,K);
  Reale = gzeros(m_dimension,K*length(Constellation_point));
  Realtemp_vector = gzeros(m_dimension,K*length(
   Constellation_point));
  Realsubscript = gzeros(1,K);

  KCondition_0 = K>length(Constellation_point);
  K1 = KCondition_0*length(Constellation_point)+(1-
   KCondition_0)*K;

  RealK_s = gdouble([]);
  for cc = 1:ConstellationSzie
      for tt = 1:K1
          Reals_h(m_dimension,tt) = Constellation_point(cc);
      end
      Y_r = Y_r-Constellation_point(cc)*HH(:,I_Real,pp);
      RealZ(:,pp) = RealQ(:,:,pp)'*Y_r;

      ii = m_dimension-1;
      Realtemp_T = gzeros(1,length(Constellation_point));
      for j = 1:length(Constellation_point)
          Realtemp_T(j) = (RealZ(ii,pp)-RealR(ii,ii,pp)*
           Constellation_point(j))^2; % Branch cost
      end
      RealSort_T = sort(Realtemp_T,'ascend');     % Sort the
       branch cost with the ascend order
      RealT(ii,1:K1) = RealSort_T(1:K1);          % Select K
       partial vectors which have the smallest PEDs
      for t = 1:K1
          Reals_h(ii,t) = Constellation_point(FindData(RealT(
```

```
     ii,t),Realtemp_T));   % save the detected nodes
end
Realtemp_s = Reals_h;

for i = m_dimension-2:-1:1
    % i-th node(i<m_dimension)
    count = 1;

    KCondition = K>(length(Constellation_point))^(
     m_dimension-i);
    K1 = KCondition*(length(Constellation_point))^(
     m_dimension-i)+(1-KCondition)*K;
    KCondition_1 = K>(length(Constellation_point))^(
     m_dimension-i+1);
    K2 = KCondition*(KCondition_1*length(
     Constellation_point)^(m_dimension-i+1)+(1-
     KCondition_1)*K)+(1-KCondition)*K;
    length_T = K1*length(Constellation_point);
    Realtemp_T = gzeros(1,length_T);
    for t=1:K1
        for j = 1:length(Constellation_point)
                         % Go through all the
          constellation nodes
            Realtemp_s(i,t) = Constellation_point(j);
            Realtemp_vector(:,count) = Realtemp_s(:,t);
            rs = 0;
            for n = i:m_dimension
                rs = rs+RealR(i,n,pp)*Realtemp_s(n,t);
                       % Calculate the branch cost for
                 each level
            end
            Reale(i,count) = (RealZ(i,pp)-rs)^2;
            Realtemp_T(count) = RealT(i+1,t)+Reale(i,
             count); % Calculate he PED
            count  = count+1;
        end
    end
    RealSort_T = sort(Realtemp_T,'ascend');  % Sort the
     branch cost with the ascend order
    RealT(i,1:K2) = RealSort_T(1:K2);        % Select K
     partial vectors which have the smallest PEDs
```

```
            for t = 1:K2                              % Pick the
             nodes retated to the partial vectors
                Realsubscript(t) = FindData(RealT(i,t),
                 Realtemp_T);
            end
            Realsubscript = sort(Realsubscript,'ascend');
            for q = 1:K2
                RealT(i,q) = Realtemp_T(Realsubscript(q));
                Reals_h(:,q) = Realtemp_vector(:,Realsubscript(q
                 ));   % Save the detected nodes and Update the
                  path
            end
            Realtemp_s = Reals_h;
        end
        RealK_s = [RealK_s Reals_h];
    end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Imagnew_H = Channel_ColumnExchange(HH(:,:,pp),
     ImagOriginal_order);
    Imagnew_H(:,m_dimension,pp) = 0;
    [ImagQ(:,:,pp),ImagR(:,:,pp)] = qr(Imagnew_H(:,:,pp));
                % QR factorization
    for k = 1:M_receive
        QRCondition = ImagR(k,k,pp)<0;
        ImagQ(:,k,pp) = (1-QRCondition)*ImagQ(:,k,pp)+
         QRCondition*ImagQ(:,k,pp)*(-1);
        ImagR(k,:,pp) = (1-QRCondition)*ImagR(k,:,pp)+
         QRCondition*ImagR(k,:,pp)*(-1);
    end
    Y_r = Y_ori;
%%%%%%%%%%%%%%%%%%%%
    ImagT = gzeros(m_dimension,K);
    Imags_h = gzeros(m_dimension,K);
    Image = gzeros(m_dimension,K*length(Constellation_point));
    Imagtemp_vector = gzeros(m_dimension,K*length(
     Constellation_point));
    Imagsubscript = gzeros(1,K);

    ImagK_s = gdouble([]);
    for cc = 1:ConstellationSzie
        for tt = 1:K1
```

```
    Imags_h(m_dimension,tt) = Constellation_point(cc);
end
Y_r = Y_r-Constellation_point(cc)*Imagnew_H(:,
 m_dimension,pp);
ImagZ(:,pp) = ImagQ(:,:,pp)'*Y_r;

ii = m_dimension-1;
KCondition_0 = K>length(Constellation_point);
K1 = KCondition_0*length(Constellation_point)+(1-
 KCondition_0)*K;
Imagtemp_T = gzeros(1,length(Constellation_point));
for j = 1:length(Constellation_point)
    Imagtemp_T(j) = (ImagZ(ii,pp)-ImagR(ii,ii,pp)*
     Constellation_point(j))^2; % Branch cost
    num_nodes = num_nodes+1;
end
ImagSort_T = sort(Imagtemp_T,'ascend'); % Sort the
 branch cost with the ascend order
ImagT(ii,1:K1) = ImagSort_T(1:K1);       % Select K
 partial vectors which have the smallest PEDs
for t = 1:K1
    Imags_h(ii,t) = Constellation_point(FindData(ImagT(
     ii,t),Imagtemp_T));   % save the detected nodes
end
Imagtemp_s = Imags_h;

for i = m_dimension-2:-1:1
    % i-th node(i<m_dimension)
    count = 1;

    KCondition = K>(length(Constellation_point))^(
     m_dimension-i);
    K1 = KCondition*(length(Constellation_point))^(
     m_dimension-i)+(1-KCondition)*K;
    KCondition_1 = K>(length(Constellation_point))^(
     m_dimension-i+1);
    K2 = KCondition*(KCondition_1*length(
     Constellation_point)^(m_dimension-i+1)+(1-
     KCondition_1)*K)+(1-KCondition)*K;
    length_T = K1*length(Constellation_point);
    Imagtemp_T = gzeros(1,length_T);
```

```
        for t=1:K1
            for j = 1:length(Constellation_point)          %
              Go through all the constellation nodes
                Imagtemp_s(i,t) = Constellation_point(j);
                Imagtemp_vector(:,count) = Imagtemp_s(:,t);
                rs = 0;
                for n = i:m_dimension
                    rs = rs+ImagR(i,n,pp)*Imagtemp_s(n,t); %
                      Calculate the branch cost for each
                      level
                end
                Image(i,count) = (ImagZ(i,pp)-rs)^2;
                num_nodes = num_nodes+1;
                Imagtemp_T(count) = ImagT(i+1,t)+Image(i,
                 count);  % Calculate he PED
                count  = count+1;
            end
        end
        ImagSort_T = sort(Imagtemp_T,'ascend');  % Sort the
         branch cost with the ascend order
        ImagT(i,1:K2) = ImagSort_T(1:K2);        % Select K
         partial vectors which have the smallest PEDs
        for t = 1:K2                             % Pick the
         nodes retated to the partial vectors
            Imagsubscript(t) = FindData(ImagT(i,t),
             Imagtemp_T);
        end
        Imagsubscript = sort(Imagsubscript,'ascend');
        for q = 1:K2
            ImagT(i,q) = Imagtemp_T(Imagsubscript(q));
            Imags_h(:,q) = Imagtemp_vector(:,Imagsubscript(q
              ));   % Save the detected nodes and Update the
              path history for each retained path
        end
        Imagtemp_s = Imags_h;
    end
    ImagK_s = [ImagK_s Imags_h];
end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Real part
    RealTempVector = Symbol_ColumnExchangeBack(RealK_s(:,:)
```

```
        .',RealOriginal_order);
        ImagTempVector = Symbol_ColumnExchangeBack(ImagK_s(:,:)
         .',ImagOriginal_order);
        s_total(:,:,pp) = [RealTempVector(:,:,pp).'
         ImagTempVector(:,:,pp).'];
        % Reach the lowest level
        b = gzeros(1,2*K*ConstellationSzie);
        for k = 1:2*K*ConstellationSzie
        b(k) = norm(Y_r-H_r*s_total(:,k,pp))^2;  % Calculate K
         PEDs
        end
        [MinSub,~] = FindMinimum(b);
        det_node = gzeros(m_dimension,NumParallel);
        det_node(:,pp) = s_total(:,MinSub,pp);       % Pick the
         vector which has the smallest PED and save it
        end
        % Symbol Errors
        for i = 1:2*M_transmit
        Count = det_node(i,pp)-S_r(i,pp);
        condition = Count~=0;
        error = error+condition;
    end
gend

SymbolError = single(sum(error));     % Cast GPU data back to
 CPU
```

B.10 Parallel VBLAST-K-Best Detection Algorithm

```
% Before the algorithm, System should be real-valued
S_r = [real(s);imag(s)];      % Generate the real version
H_r = [real(H) -1*imag(H);imag(H) real(H)];
noise_r = [real(noise);imag(noise)];

% Choose the value of K
% Choose the number K_Layer to execute K-Best

function SymbolError = VBLAST_K(M_transmit,M_receive,m_dimension
 ,K,H_r,Y_r,K_Layer,snr,Constellation_point,partition)

% Input:  M_transmit: number of transmit antennas
```

```
%          M_receive: number of received antennas
%          m_dimension: search level
%          K: the number of the selected best node
%          H_r: real-valued of the Channel matrix
%          Y_r: real-valued of the received signal y
%          K_Layer: number of layers to be executed with K-Best
%          snr: signal-to-noise ratio
%          Constellation_point: real-valued constellation
%          partition: constellation points partition
% output: SymbolError: the matrix of the detected node at each
 level

global pp
global NumParallel
error = gzeros(NumParallel,1);

V_BLAST_Layer = m_dimension-K_Layer;
num_nodes = 0;
ConstellationSzie = length(Constellation_point);

IdentityMat = geye(2*M_transmit);

gfor pp = 1:NumParallel
        HH(:,:,pp) = H_r;
    RealTempH(:,:,pp) = H_r;
    ImagTempH(:,:,pp) = H_r;

        transpose_h(:,:,pp) = HH(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*HH(:,:,pp)+(1/
         snr)*IdentityMat;
gend
AfterInverseMat = NewInverse(InverseMat);
gfor pp = 1:NumParallel
        Vblast_Q(:,:,pp) = AfterInverseMat(:,:,pp)*transpose_h
         (:,:,pp);
gend

    G = Vblast_Q;
    RealG = gzeros(2*M_transmit,2*M_receive,2*M_transmit,
     NumParallel);
    ImagG = gzeros(2*M_transmit,2*M_receive,2*M_transmit,
```

```
    NumParallel);
    NormQ = gzeros(1,M_transmit);
    RealNormQ = gzeros(1,2*M_transmit);
    ImagNormQ = gzeros(1,2*M_transmit);

gfor pp = 1:NumParallel
        for p = 1:M_transmit
                NormQ(p) = (norm(G(p,:,pp)))^2;  % calculate the
                  normal value of G
        end
        [~,I_Real] = max(NormQ);             % I is the
         subscript of the maximum value
        RealTempH(:,I_Real,pp) = 0;
        transpose_h(:,:,pp) = RealTempH(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*RealTempH(:,:,
         pp)+(1/snr)*IdentityMat;
gend
AfterInverseMat = NewInverse(InverseMat);
gfor pp = 1:NumParallel
        RealG(:,:,1,pp) = AfterInverseMat(:,:,pp)*transpose_h
         (:,:,pp);

    for qq = 1:2*M_transmit
        RealNormQ(qq) = (norm(RealG(qq,:,1,pp)))^2; % calculate
         the normal value of G
    end
    I_Imag = I_Real+ConstellationSzie;
    ImagTempH(:,I_Imag,pp) = 0;

        transpose_h(:,:,pp) = ImagTempH(:,:,pp)';
        InverseMat(:,:,pp) = transpose_h(:,:,pp)*ImagTempH(:,:,
         pp)+(1/snr)*IdentityMat;
gend
AfterInverseMat = NewInverse(InverseMat);
gfor pp = 1:NumParallel
        ImagG(:,:,1,pp) = AfterInverseMat(:,:,pp)*transpose_h
         (:,:,pp);

    for qq = 1:2*M_transmit
        ImagNormQ(qq) = (norm(ImagG(qq,:,1,pp)))^2; % calculate
         the normal value of G
```

```
    end
gend
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Real part Norm G
RealVBLASTk = gzeros(1,2*M_transmit-1);
for jj = 1:2*M_transmit-1
    gfor pp = 1:NumParallel
        RealNormQ(I_Real) = Inf;
        for t = 1:jj-1
            RealNormQ(RealVBLASTk(t)) = Inf;  % set the detected
              normal value to infinity
        end
        [~,Real] = min(RealNormQ);            % I is the
         subscript of the minimum value
        RealVBLASTk(jj) = Real;
        RealTempH(:,RealVBLASTk(jj),pp) = 0;

                transpose_h(:,:,pp) = RealTempH(:,:,pp)';
                InverseMat(:,:,pp) = transpose_h(:,:,pp)*
                 RealTempH(:,:,pp)+(1/snr)*IdentityMat;
        gend
        AfterInverseMat = NewInverse(InverseMat);
        gfor pp = 1:NumParallel
                RealG(:,:,jj+1,pp) = AfterInverseMat(:,:,pp)*
                 transpose_h(:,:,pp);
        for qq = 1:length(RealTempH(:,1))
            RealNormQ(qq) = (norm(RealG(qq,:,jj+1,pp)))^2;  %
             calculate the normal value of G
        end
    gend
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Imaginary part Norm G
ImagVBLASTk = gzeros(1,2*M_transmit-1);
for jj = 1:2*M_transmit-1
        gfor pp = 1:NumParallel
        ImagNormQ(I_Imag) = Inf;
        for t = 1:jj-1
            ImagNormQ(ImagVBLASTk(t)) = Inf;  % set the detected
              normal value to infinity
        end
```

```
        [~,Imag] = min(ImagNormQ);            % I is the
         subscript of the minimum value
        ImagVBLASTk(jj) = Imag;
        ImagTempH(:,ImagVBLASTk(jj),pp) = 0;

                transpose_h(:,:,pp) = ImagTempH(:,:,pp)';
                InverseMat(:,:,pp) = transpose_h(:,:,pp)*
                 ImagTempH(:,:,pp)+(1/snr)*IdentityMat;
        gend
        AfterInverseMat = NewInverse(InverseMat);
        gfor pp = 1:NumParallel
                ImagG(:,:,jj+1,pp) = AfterInverseMat(:,:,pp)*
                 transpose_h(:,:,pp);
        for qq = 1:length(ImagTempH(:,1))
            ImagNormQ(qq) = (norm(ImagG(qq,:,jj+1,pp)))^2; %
             calculate the normal value of G
        end
    gend
end
gfor pp = 1:NumParallel
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Real part V-BLAST
    RealSymbolTestRow = gzeros(ConstellationSzie,V_BLAST_Layer,
     NumParallel);
    RealVBLASTSymbolTest = gzeros(V_BLAST_Layer-1,1,NumParallel)
     ;
    for tt = 1:ConstellationSzie
        TestY = gzeros(m_dimension,NumParallel);
        TestY(:,pp) = Y_r;
        TestH = HH;
        TestY(:,pp) = TestY(:,pp)-Constellation_point(tt)*TestH
         (:,I_Real,pp);
        TestH(:,I_Real,pp) = 0;
        for jj = 1:V_BLAST_Layer-1
            shk = gzeros(1,NumParallel);
            % nulling
            shk(:,pp) = RealG(RealVBLASTk(jj),:,jj,pp)*TestY(:,
             pp);
                % Slicing
            [~,RealValue] = quantization(shk(:,pp),partition,
             Constellation_point);
```

```
            RealVBLASTSymbolTest(jj,1,pp) = RealValue;    % get
             the real detected symbol
                % interference cancellation
            TestY(:,pp) = TestY(:,pp)-RealValue*TestH(:,
             RealVBLASTk(jj),pp);
            TestH(:,RealVBLASTk(jj),pp) = 0;   % set the used
             channel into 0
        end
        RealSymbolTestRow(tt,:,pp) = [Constellation_point(tt)
         RealVBLASTSymbolTest(:,1,pp).'];
    end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Imaginary part V-BLAST
    ImagSymbolTestRow = gzeros(ConstellationSzie,V_BLAST_Layer,
     NumParallel);
    ImagVBLASTSymbolTest = gzeros(V_BLAST_Layer-1,NumParallel);
    for tt = 1:ConstellationSzie
        TestY = gzeros(m_dimension,NumParallel);
        TestY(:,pp) = Y_r;
        TestH = HH;
        TestY(:,pp) = TestY(:,pp)-Constellation_point(tt)*TestH
         (:,I_Imag,pp);
        TestH(:,I_Imag,pp) = 0;
        for jj = 1:V_BLAST_Layer-1
            shk = gzeros(1,NumParallel);
                        % nulling
            shk(:,pp) = ImagG(ImagVBLASTk(jj),:,jj,pp)*TestY(:,
             pp);
                % Slicing
            [~,ImagValue] = quantization(shk(:,pp),partition,
             Constellation_point);
            ImagVBLASTSymbolTest(jj,pp) = ImagValue;       % get
              the real detected symbol
                % interference cancellation
            TestY(:,pp) = TestY(:,pp)-ImagValue*TestH(:,
             ImagVBLASTk(jj),pp);
            TestH(:,ImagVBLASTk(jj),pp) = 0;     % set the used
             channel into 0
        end
        ImagSymbolTestRow(tt,:,pp) = [Constellation_point(tt)
         ImagVBLASTSymbolTest(:,pp).'];
```

```
    end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Realorder = [I_Real RealVBLASTk(1:V_BLAST_Layer-1)];
    Imagorder = [I_Imag ImagVBLASTk(1:V_BLAST_Layer-1)];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Realorder_count = 1;
    Realnew_order = gzeros(1,K_Layer);
    for i = m_dimension-1:-1:V_BLAST_Layer
        Realnew_order(Realorder_count) = RealVBLASTk(i);
        Realorder_count = Realorder_count+1;
    end
%%%%%%%%%%%%%%%%%%%%
    Realtotal_order = [Realorder Realnew_order];
    Realnew_H = Channel_ColumnExchange(HH(:,:,pp),
     Realtotal_order);
    [RealQ(:,:,pp),RealR(:,:,pp)] = qr(Realnew_H(:,:,pp));   %
     QR factorization
    for k = 1:length(Y_r(:,1))
        QRCondition = RealR(k,k,pp)<0;
        RealQ(:,k,pp) = (1-QRCondition)*RealQ(:,k,pp)+
         QRCondition*RealQ(:,k,pp)*(-1);
        RealR(k,:,pp) = (1-QRCondition)*RealR(k,:,pp)+
         QRCondition*RealR(k,:,pp)*(-1);
    end
    RealZ(:,pp) = RealQ(:,:,pp)'*Y_r;
%%%%%%%%%%%%%%%%%%%%
        % Real part K-Best
    RealT = gzeros(m_dimension,K);
    Reals_h = gzeros(m_dimension,K);
    Reale = gzeros(m_dimension,K*length(Constellation_point));
    Realtemp_vector = gzeros(m_dimension,K*length(
     Constellation_point));
    Realsubscript = gzeros(1,K);

    i = m_dimension;
    KCondition_0 = K>length(Constellation_point);
    K1 = KCondition_0*length(Constellation_point)+(1-
     KCondition_0)*K;
    Realtemp_T = gzeros(1,length(Constellation_point));
    for j = 1:length(Constellation_point)
        Realtemp_T(j) = (RealZ(i,pp)-RealR(i,i,pp)*
```

```
     Constellation_point(j))^2; % Branch cost
    num_nodes = num_nodes+1;
end
RealSort_T = sort(Realtemp_T,'ascend'); % Sort the branch
 cost with the ascend order
RealT(i,1:K1) = RealSort_T(1:K1);       % Select K partial
 vectors which have the smallest PEDs
for t = 1:K1
    Reals_h(i,t) = Constellation_point(FindData(RealT(i,t),
     Realtemp_T));   % save the detected nodes
end
Realtemp_s = Reals_h;

for i = m_dimension-1:-1:m_dimension-K_Layer+1
    % i-th node(i<m_dimension)
    count = 1;

    KCondition = K>(length(Constellation_point))^(
     m_dimension-i);
    K1 = KCondition*(length(Constellation_point))^(
     m_dimension-i)+(1-KCondition)*K;
    KCondition_1 = K>(length(Constellation_point))^(
     m_dimension-i+1);
    K2 = KCondition*(KCondition_1*length(Constellation_point
     )^(m_dimension-i+1)+(1-KCondition_1)*K)+(1-KCondition)
     *K;
    length_T = K1*length(Constellation_point);
    Realtemp_T = gzeros(1,length_T);
    for t=1:K1
        for j = 1:length(Constellation_point)
                       % Go through all the constellation
         nodes
            Realtemp_s(i,t) = Constellation_point(j);
            Realtemp_vector(:,count) = Realtemp_s(:,t);
            rs = 0;
            for n = i:m_dimension
                rs = rs+RealR(i,n,pp)*Realtemp_s(n,t);
                      % Calculate the branch cost for each
                   level
            end
            Reale(i,count) = (RealZ(i,pp)-rs)^2;
```

```
            num_nodes = num_nodes+1;
            Realtemp_T(count) = RealT(i+1,t)+Reale(i,count);
              % Calculate he PED
            count  = count+1;
        end
    end
    RealSort_T = sort(Realtemp_T,'ascend');   % Sort the
     branch cost with the ascend order
    RealT(i,1:K2) = RealSort_T(1:K2);         % Select K
     partial vectors which have the smallest PEDs
    for t = 1:K2                              % Pick the
     nodes retated to the partial vectors
        Realsubscript(t) = FindData(RealT(i,t),Realtemp_T);
    end
    Realsubscript = sort(Realsubscript,'ascend');
    for q = 1:K2
        RealT(i,q) = Realtemp_T(Realsubscript(q));
        Reals_h(:,q) = Realtemp_vector(:,Realsubscript(q));
            % Save the detected nodes and Update the path
    end
    Realtemp_s = Reals_h;
end
RealK_s(:,:,pp) = Reals_h(V_BLAST_Layer+1:m_dimension,:);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Imagorder_count = 1;
Imagnew_order = gzeros(1,K_Layer);
for i = m_dimension-1:-1:V_BLAST_Layer
    Imagnew_order(Imagorder_count) = ImagVBLASTk(i);
    Imagorder_count = Imagorder_count+1;
end
%%%%%%%%%%%%%%%%%%%%
    Imagtotal_order = [Imagorder Imagnew_order];
    Imagnew_H = Channel_ColumnExchange(HH(:,:,pp),
     Imagtotal_order);
    [ImagQ(:,:,pp),ImagR(:,:,pp)] = qr(Imagnew_H(:,:,pp));
            % QR factorization
    for k = 1:length(Y_r(:,1))
    QRCondition = ImagR(k,k,pp)<0;
    ImagQ(:,k,pp) = (1-QRCondition)*ImagQ(:,k,pp)+
     QRCondition*ImagQ(:,k,pp)*(-1);
    ImagR(k,:,pp) = (1-QRCondition)*ImagR(k,:,pp)+
```

```
 QRCondition*ImagR(k,:,pp)*(-1);
end
ImagZ(:,pp) = ImagQ(:,:,pp)'*Y_r;
%%%%%%%%%%%%%%%%%%%%
% Imaginary part K-Best
ImagT = gzeros(m_dimension,K);
Imags_h = gzeros(m_dimension,K);
Image = gzeros(m_dimension,K*length(Constellation_point)
 );
Imagtemp_vector = gzeros(m_dimension,K*length(
 Constellation_point));
Imagsubscript = gzeros(1,K);

i = m_dimension;
KCondition_0 = K>length(Constellation_point);
K1 = KCondition_0*length(Constellation_point)+(1-
 KCondition_0)*K;
Imagtemp_T = gzeros(1,length(Constellation_point));
for j = 1:length(Constellation_point)
Imagtemp_T(j) = (ImagZ(i,pp)-ImagR(i,i,pp)*
 Constellation_point(j))^2; % Branch cost
num_nodes = num_nodes+1;
end
ImagSort_T = sort(Imagtemp_T,'ascend');    % Sort the
 branch cost with the ascend order
ImagT(i,1:K1) = ImagSort_T(1:K1);          % Select K
 partial vectors which have the smallest PEDs
for t = 1:K1
Imags_h(i,t) = Constellation_point(FindData(ImagT(i,t),
 Imagtemp_T));  % save the detected nodes
end
Imagtemp_s = Imags_h;

for i = m_dimension-1:-1:m_dimension-K_Layer+1
% i-th node(i<m_dimension)
count = 1;

KCondition = K>(length(Constellation_point))^(
 m_dimension-i);
K1 = KCondition*(length(Constellation_point))^(
 m_dimension-i)+(1-KCondition)*K;
```

```
KCondition_1 = K>(length(Constellation_point))^(
 m_dimension-i+1);
K2 = KCondition*(KCondition_1*length(Constellation_point
 )^(m_dimension-i+1)+(1-KCondition_1)*K)+(1-KCondition)
 *K;
length_T = K1*length(Constellation_point);
Imagtemp_T = gzeros(1,length_T);
for t=1:K1
    for j = 1:length(Constellation_point)           % Go
       through all the constellation nodes
        Imagtemp_s(i,t) = Constellation_point(j);
        Imagtemp_vector(:,count) = Imagtemp_s(:,t);
        rs = 0;
        for n = i:m_dimension
            rs = rs+ImagR(i,n,pp)*Imagtemp_s(n,t);  %
             Calculate the branch cost for each level
        end
        Image(i,count) = (ImagZ(i,pp)-rs)^2;
        num_nodes = num_nodes+1;
        Imagtemp_T(count) = ImagT(i+1,t)+Image(i,count);
           % Calculate he PED
        count  = count+1;
    end
end
ImagSort_T = sort(Imagtemp_T,'ascend');       % Sort the
  branch cost with the ascend order
ImagT(i,1:K2) = ImagSort_T(1:K2);             % Select K
  partial vectors which have the smallest PEDs
for t = 1:K2                                  % Pick the
  nodes retated to the partial vectors
   Imagsubscript(t) = FindData(ImagT(i,t),Imagtemp_T);
end
Imagsubscript = sort(Imagsubscript,'ascend');
for q = 1:K2
    ImagT(i,q) = Imagtemp_T(Imagsubscript(q));
    Imags_h(:,q) = Imagtemp_vector(:,Imagsubscript(q));
       % Save the detected nodes and Update the path
end
Imagtemp_s = Imags_h;
end
ImagK_s(:,:,pp) = Imags_h(V_BLAST_Layer+1:m_dimension,:)
```

```
    ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Real part total Symbol
        Realtemp_s_total = gzeros(m_dimension,K*
         ConstellationSzie,NumParallel);
        RealTotalSymbol = 0;
        for ll = 1:ConstellationSzie
        for vv = 1:K
                RealTotalSymbol = RealTotalSymbol+1;
                Realtemp_s_total(RealTotalSymbol,:,pp) = [
                 RealSymbolTestRow(ll,:,pp) RealK_s(:,vv,pp)
                 .'];
        end
        end
        % Imaginary part total Symbol
        Imagtemp_s_total = gzeros(m_dimension,K*
         ConstellationSzie,NumParallel);
        ImagTotalSymbol = 0;
        for ll = 1:ConstellationSzie
        for vv = 1:K
                ImagTotalSymbol = ImagTotalSymbol+1;
                Imagtemp_s_total(ImagTotalSymbol,:,pp) = [
                 ImagSymbolTestRow(ll,:,pp) ImagK_s(:,vv,pp)
                 .'];
        end
        end

        RealTempVector = Symbol_ColumnExchangeBack(
         Realtemp_s_total(:,:,pp),Realtotal_order);
        ImagTempVector = Symbol_ColumnExchangeBack(
         Imagtemp_s_total(:,:,pp),Imagtotal_order);
        s_total(:,:,pp) = [RealTempVector(:,:,pp).'
         ImagTempVector(:,:,pp).'];
        % Reach the lowest level
        b = gzeros(1,2*K*ConstellationSzie);
        for k = 1:2*K*ConstellationSzie
        b(k) = norm(Y_r-H_r*s_total(:,k,pp))^2;   % Calculate K
         PEDs
        end
        [MinSub,~] = FindMinimum(b);
        det_node = gzeros(m_dimension,NumParallel);
```

```
        det_node(:,pp) = s_total(:,MinSub,pp);        % Pick the
          vector which has the smallest PED and save it
gend

SymbolError = single(sum(error));     % Cast GPU data back to
 CPU
```

∼