PARALLELIZATION OF NOISE SUBSPACE-BASED DOA ESTIMATION ALGORITHMS ON CPU AND GPU

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HAMZA ERAY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
MODELLING AND SIMULATION

FEBRUARY 2021

Approval of the thesis:

PARALLELIZATION OF NOISE SUBSPACE-BASED DOA ESTIMATION ALGORITHMS ON CPU AND GPU

submitted by **HAMZA ERAY** in partial fulfillment of the requirements for the degree of **Master of Science in Modelling and Simulation Department, Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek BOZŞAHİN
Dean, Graduate School of **Informatics** _____

Assist. Prof. Dr. Elif SÜRER
Head of Department, **Modelling and Simulation, METU** _____

Prof. Dr. Alptekin TEMİZEL
Supervisor, **Modelling and Simulation, METU** _____

Examining Committee Members:

Date: .....

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: HAMZA ERAY

Signature :

ABSTRACT

PARALLELIZATION OF NOISE SUBSPACE-BASED DOA ESTIMATION ALGORITHMS ON CPU AND GPU

Eray, Hamza

M.S., Department of Modelling and Simulation

Supervisor: Prof. Dr. Alptekin TEMİZEL

February 2021, 76 pages

Direction-of-Arrival (DOA) estimation is known as an active research area, and it is studied under array signal processing. The algorithms in this area are widely used in various applications such as sonar, search-and-rescue, navigation, and geolocation. However, achieving a real-time system performance is sometimes a challenging task for these algorithms.

In this thesis, four noise subspace-based DOA estimation algorithms (PHD, MUSIC, EV, and MN) were considered and implemented in MATLAB, C/C++, and CUDA. MATLAB implementations were mainly used for theoretical and numerical analyses. Whereas, C/C++ implementations were initially used for constructing the parallelization structure and they were realized in two versions working serially and in parallel manner (via OpenMP).

Within the scope of theoretical analysis, these algorithms were compared with each other in terms of DOA estimation accuracy and effects of change in different parameters (e.g., array geometry, array aperture, SNR level, etc.) on the accuracy were observed. On the other hand, in terms of implementation-based experiments, all the codes in MATLAB, C/C++, and CUDA were evaluated from both numerical and performance viewpoints. The general numerical validation of C/C++ and CUDA codes was realized against ground-truth MATLAB codes. After the initial assessment of the CUDA code performance, some GPU-based optimizations were applied and the corresponding performance improvements were evaluated. The CUDA code performance was benchmarked on the PC platforms with different system configurations.

Consequently, a considerable speedup was achieved for CUDA codes compared to baseline multi-threaded C/C++ codes.

Keywords: CUDA, direction-of-arrival estimation, GPU, OpenMP, parallelization

ÖZ

GÜRÜLTÜ ALTUZAYI TABANLI DOA KESTİRİM ALGORİTMALARININ CPU VE GPU ÜZERİNDE PARALLELLEŞTİRİLMESİ

Eray, Hamza

Yüksek Lisans, Modelleme ve Simülasyon Anabilim Dalı Bölümü

Tez Yöneticisi : Prof. Dr. Alptekin TEMİZEL

Şubat 2021 , 76 sayfa

Varış Yönü (DOA) kestirimi, aktif bir araştırma alanı olarak bilinir ve dizi sinyal işleme altında incelenir. Bu alandaki algoritmalar; sonar, arama-kurtarma, navigasyon ve konum belirleme gibi çeşitli uygulamalarda yaygın olarak kullanılmaktadır. Ancak, gerçek zamanlı bir sistem performansı elde etmek bazen bu algoritmalar için zorlu bir hale gelmektedir.

Bu tezde, gürültü altuzayı-tabanlı dört farklı DOA kestirim algoritması (PHD, MUSIC, EV ve MN) ele alınmış ve her biri MATLAB, C/C++ ve CUDA'da gerçeklenmiştir. MATLAB gerçeklemeleri, ağırlıklı olarak teorik ve nümerik analizler için kullanılmıştır. C/C++ gerçeklemeleri ise algoritmaların paralelleştirme yapısını oluşturmak için kullanılmış ve her birinin seri ve (OpenMP ile) paralel olarak çalışan iki versiyonu gerçeklenmiştir.

Teorik analizler kapsamında, bu algoritmalar DOA kestirim doğruluğu açısından birbirleriyle kıyaslanmış ve farklı parametrelerdeki (örn. dizi geometrisi, dizi açıklığı, SNR düzeyi vb.) değişimin doğruluk üzerindeki etkileri gözlemlenmiştir. Öte yandan uygulamaya dayalı deneylerde ise, MATLAB, C/C++ ve CUDA'daki tüm kodlar hem nümerik hem de performans açısından incelenmiştir. C/C++ ve CUDA kodlarının genel nümerik doğrulaması, (duyarlığına güvenilen) MATLAB kodlarına karşı gerçekleştirilmiştir. CUDA kod performansının ilk değerlendirmesinden sonra, GPU-tabanlı birtakım optimizasyonlar uygulanmış ve performanstaki ilgili iyileşmeler değerlendirilmiştir. CUDA kod performansı, farklı sistem konfigürasyonlarına sahip PC platformlarında karşılaştırmalı olarak analiz edilmiştir. Sonuç olarak; çoklu iş-parçacıklı temel C/C++ kodlarına kıyasla, CUDA kodları ile performansta kayda değer bir hız-

lanma sağlanmıştır.

Anahtar Kelimeler: CUDA, varış yönü kestirme, GPU, OpenMP, paralelleştirme

Dedicated to all the people who add value to my life and
to the eternal memory of my nephew, Çağatay

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere and deep gratitude to my advisor Prof. Dr. Alptekin TEMİZEL for his great support and encouragement. His strong knowledge and kind guidance made it possible to complete my thesis period.

Secondly, I would like to thank all my professors in the jury for attending to my thesis defense and expressing their opinion on that.

Thanks to all my colleagues from ESEN Sistem Entegrasyon due to their technical and motivational supports. Special thanks to Dr. Emrah ONAT for allowing me to use the synthetic signal generator code in my thesis.

I would like to thank all my friends who are around me and make my life enjoyable.

Another special thanks to my dear Irmak, who makes me always feel valued and helps my motivation rise.

Finally, I would like to thank my mother, father and sister for being always supportive in my life, and thanks also to my cute nephew for boosting my mood.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

1D	1-Dimensional
2D	2-Dimensional
3D	3-Dimensional
AIC	Akaike Information Criterion
API	Application Programming Interface
CORDIC	COordinate Rotation DIgital Computer
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DOA	Direction of Arrival
DF	Direction Finding
DFT	Discrete Fourier Transform
DP	Direct Path (of a propagating signal)
DSP	Digital Signal Processing
EM	Electromagnetic
EV	Eigen Vector
EVD	Eigenvalue Decomposition
FFT	Fast Fourier Transform
FP32	32-bit floating-point (single-precision)
FP64	64-bit floating-point (double-precision)
FPGA	Field Programmable Gate Array
GPGPU	General-Purpose GPU
GPU	Graphics Processing Unit
HF	High Frequency
MN	Minimum Norm

MUSIC	MUltiple SIgnal Classification
MVDR	Minimum Variance Distortionless Response
MP	Multipath (of a propagating signal)
OpenMP	Open Multi-Processing
OS	Operating System
PHD	Pisarenko Harmonic Decomposition
PHWT	Parallel Haar Wavelet Transform
RF	Radio Frequency
SIMD	Single Instruction, Multiple Data
SNR	Signal-to-Noise Ratio
SPMD	Single Program, Multiple Data
SVD	Singular Value Decomposition
SWaP	Size, Weight, and Power
UCA	Uniform Circular Array
ULA	Uniform Linear Array

CHAPTER 1

INTRODUCTION

1.1 Problem Definition

The direction of arrival (DOA) estimation is the process of determining the direction of signals which can be possibly in the form of electromagnetic (EM), acoustic/sonic or seismic waves [1]. This process is generally achieved by a certain type of sensor arrays such as antenna, microphone/hydrophone or geophone array.

The DOA estimation is known as a key research area in several civilian and military applications. Some examples for these applications are navigation, geolocation, radar, search-and-rescue, sonar and seismology [2, 3]. In the literature, it is generally studied under array signal processing. Studies on DOA estimation date back to the early 1900s (WWI) with the invention of Radio Direction Finder (RDF) systems by Bellini and Tosi [4]. Several advances have been achieved from this date in concepts/principles and more importantly in the technology [5]. With the help of these, a strong theoretical and practical basis for DOA estimation has been constructed until today. Nevertheless, it is still an ever-developing and quite active research area with proposed novel approaches [2].

As will be discussed in more detail in Section 2.2, there are various DOA estimation methods in the literature. Within the scope of this thesis, DOA estimation for the EM (radio) signals was considered in the narrowband scenario and it was focused on four noise subspace-based DOA estimation algorithms:

- Pisarenko Harmonic Decomposition (PHD)
- MUltiple SIgnal Classification (MUSIC)
- EigenVector (EV)
- Minimum-Norm (MN)

As long as sufficient amount of data is fed and SNR of the signal is at an acceptable level, these algorithms mentioned above (especially MUSIC) are known for providing convergent estimates (i.e., statistically consistent) and higher angular resolution, in contrast to the some beamforming methods like Capon [6]. However, some parts of these algorithms are mathematically intensive and in some cases, it may require substantial computing power for their real-time implementations [7]. Therefore, there

was a need for having numerically stable and faster implementations for these algorithms following some parallelization approaches.

1.2 Related Studies

MUSIC algorithm is the basis for the noise subspace DOA methods and one of the most popular DOA algorithms. Hence, parallelization-related studies mainly focus on the MUSIC algorithm among the four noise subspace-based methods mentioned. Since field programmable gate array (FPGA) was a popular parallel hardware solution for many problems before the general purpose usage of GPU technology, early works related to the MUSIC implementation are on FPGAs. After the emergence of GPGPU technology, few studies were made possible which mainly focus on the GPU implementation of the MUSIC algorithm for the DOA estimation of wideband/broadband signals. All these studies are summarized in the chronological order as follows:

- In an early work by Kim et al. [8], the spectral unitary MUSIC algorithm for the DOA estimation of narrowband signals is implemented on FPGA that computes all DSP functions by only fixed-point operation with finite bit-length, preferably. High performance in eigenvalue decomposition (EVD) and angular spectra computation are achieved by the existing system via CORDIC(COordinate Rotation DIgital Computer)-based cyclic Jacobi processor and spatial DFT operation, respectively. Even if the performance is highly satisfactory, the estimation accuracy is sometimes negatively affected by FPGA-specific fast processing factors like finite bit-length and bit-truncation.
- In a work by Zou et al. [9], an improved MUSIC algorithm is proposed with a high-speed parallel implementation working fully on FPGA. In this study, to avoid the eigenstructure decomposition of the full covariance matrix, an optimization algorithm on FPGA is proposed where the signal subspace is acquired by using a submatrix of the full covariance matrix. Estimating the covariance matrix and spectral peak search are also realized on FPGA. The computational cost of this optimization algorithm is significantly lower compared to the original MUSIC algorithm, hence this FPGA implementation performs four times faster than the DSP (TM320C6711) counterpart. Even if the implementation proposed in this work reduces the computational complexity, some differences with the original algorithm may cause a performance degradation in the DOA estimation.
- In a work by Majid et al. [7], the wideband MUSIC algorithm for DOA estimation is implemented on multi-core CPU, GPU, and IBM Cell BE processor for real-time applications. This algorithm divides the wide frequency band into narrowband components and a focusing matrix is constructed using the initial DOA estimate. This focusing matrix is used as a new covariance matrix and after QR decomposition, Akaike Information Criterion (AIC) is applied in order to determine the number of detected sources and realize the separation of signal and noise subspaces. The parallel GPU implementation (NVIDIA GTX 260) is realized for all major parallelizable parts including FFT, estimating covariance matrix, householder transformation, QR decomposition, AIC and power

method. An adaptive load balancing algorithm (ALBA) is utilized for preserving the efficiency of the implementation in any size of signal. The conclusion of the study is that the GPU implementation outperforms multi-core CPU and IBM Cell BE implementations. Another important point from the results is that there should be a balance between the number of tasks and data size in order to reduce the overhead and idle state of the processors.

- In another work by Majid et al. [10], the wideband MUSIC algorithm is implemented on multi-core CPU and GPU. In this study, the parallel steps of the implementation include the Parallel Haar Wavelet Transform (PHWT) instead of parallel FFT. Remaining all steps are similar to the previous one. The PHWT-based wideband DOA estimation generally provides availability for non-uniform array and non-stationary signal cases. The performance experiments show that both implementations (multi-core and GPU) of PHWT-based parallel MUSIC algorithm have a reduced computation time compared to the previous versions of parallel MUSIC algorithm.

- In a recent work by Lu et al. [11], parallel MUSIC algorithm is implemented on a GPU platform (Tesla K80) for the DOA estimation of broadband underwater acoustic signals. In this implementation, the broadband underwater signal is divided into several narrowband frequency bands via FFT method and each one is processed to get narrowband power spectrum. In order to determine the computational load distribution and obtain a baseline for GPU implementation, a hotspot analysis is done by realizing the serial MUSIC code on the CPU. This analysis shows that the majority of the overall execution time belongs to the azimuth spectrum computation. EVD is still computed on the CPU and the azimuth spectrum (major hotspot) is computed on the GPU utilizing an efficiently integrated usage of global and shared memory units. Different experiments are realized to compare the speedups under different system parameters. Up to 23x speedup is reported at 256 array elements and 1000 frequency points.

1.3 Motivation and Proposed Approach

As mentioned in Section 1.1, in order to obtain a satisfactory time performance for the noise subspace-based DOA methods, some compute-intensive parts require special attention from the point of implementation methodology. As listed in Section 1.2, there are some studies overcoming the computational burden via parallel hardware solutions using FPGA. Even if some of them provide an expected performance, these implementations are susceptible to a precision problem and hence they have a reduced estimation accuracy in some cases. Apart from this, as long as some special requirements like low SWaP (size, weight, and power) do not exist, the implementation on a consumer-level computer can provide a better solution instead of a special FPGA hardware and more troublesome code development cycle.

With the advent of important developments in the GPU technology from both hardware and software point of view, general-purpose computation on GPU (GPGPU) has become popular in various areas such as scientific computing, robotics, computer visualization and so on [12]. Array signal processing is also a research area where

GPGPU technology is actively used [13], [14], [15].

To this respect, faster implementation of noise subspace DOA methods utilizing the GPGPU is the main motivation of this thesis. In order to realize the GPU implementations, appropriate parts within these DOA methods have been parallelized and optimized using CUDA platform of NVIDIA©.

To be able to measure the relative numerical accuracy and time performance of the CUDA codes, these methods were firstly implemented in MATLAB and C/C++ (serial and multi-core). By doing this, various effects including different precision levels (FP32 and FP64) were investigated from the numerical perspective. Moreover, from the performance-based perspective, these baseline implementations enabled us to observe the effects of using different programming models and architectures.

To have a more holistic view, before implementing all the methods in C/C++ and CUDA, their DOA estimation performances were evaluated under different DOA parameters (array geometry, aperture, and SNR) in MATLAB.

1.4 Contributions and Novelties

The major contributions presented within the scope of this thesis are as follows:

- The DOA estimation methods named as PHD, EV and MN have not been implemented before on GPU using CUDA.
- There are some studies on the GPU implementation of broadband MUSIC algorithm, but, to the best of our knowledge, there is no GPU implementation of the narrowband case in the literature.
- A comprehensive optimization on GPU has been done, guided by the computational analysis and benchmarking of the algorithms.
- The scalability of these implementations were investigated via measuring the code performances on different hardware configurations.

Regarding the work done in this thesis, a paper was presented in 2020 High Performance Computing Conference (BASARIM) [16].

Moreover, the source codes for the CUDA implementations of the algorithms were made publicly accessible via the following GitHub link:

`https://github.com/erayhamza/NssDOACuda`

1.5 The Outline of the Thesis

The structure of the thesis can be summarized as follows:

In Chapter 2, firstly, some background information is given about the DOA estimation including its practical applications, brief history and classification. Before mentioning all the noise subspace-based DOA methods under our consideration, a system model is presented with all details (definitions, assumptions and data model).

In Chapter 3, two main parallelization environments used in this thesis, OpenMP and CUDA are introduced including their programming, execution and memory models.

In Chapter 4, all methods are firstly compared via their pseudocodes. Then, all the implementations of our methods in MATLAB, C/C++ and CUDA are presented with the relevant details.

In Chapter 5, the experiments and the corresponding results are presented in two main categories based on DOA estimation accuracy and on the numerical and performance-related issues. In the first category, DOA estimation analyses are done under different DOA parameters. In the second category, all the analyses based on numerical validation and performance are presented.

In Chapter 6, all works done throughout the thesis are mentioned briefly and the main output of this thesis work is presented. Finally, some future works that can be followed are mentioned.

CHAPTER 2

DOA ESTIMATION

From the EM wave point of view, Direction-of-Arrival (DOA) estimation is the direction information retrieval process for EM wave emitting sources, and this process is generally applied to the output signals collected from an antenna array, possibly after some preprocessing steps.

In this chapter, firstly, some practical applications are presented. Afterwards, a brief history and the classification of DOA estimation methods are mentioned. Then the data model is presented with the details on the problem parameters and related assumptions. Finally, all noise subspace-based algorithms within the scope of this thesis are explained.

2.1 Practical Applications

There are numerous practical applications of DOA estimation in various areas. Since EM (radio) perspective for DOA estimation is assumed in this thesis, some examples based on radio signals only are presented here:

- **Radio Navigation:** One of the well-known examples is LORAN (Long Range Navigation) and it is a position finding system for maritime and aeronautical navigation, by receiving typical radio signals from the stations of well-known position. Unlike radar, LORAN operates at much lower frequencies (about 2 MHz) and this enables its waves to travel not only over the surface of the earth, but also to hundred of miles from the transmitter by being reflected from the ionosphere [17]. They are generally used for helping aircrafts or ships to find their desired directions and locations.
- **Search and rescue services:** There are some international search-and-rescue initiatives like COSPAS-SARSAT which consists of 45 participants (by 2019) including Turkey as a member at ground segment provider status. The main purpose of the system in this initiative is to detect the emergency distress signals from the registered radio beacons activated by ships, aircraft or just persons and determine their corresponding locations [18].
- **Signal DF and location systems:** These systems are generally designed to operate along the radio spectrum from HF band (3-30 MHz) up to 18 GHz. They are used for various purposes that vary from wildlife tracking to intelligence gathering [19].

- **Radio astronomy:** The interferometric systems used for radio astronomy consist of large dish radio antennas and their continent-sized arrays such that the inter-element spacing in such systems can reach up to thousands of miles. Such large arrays with interferometry techniques provide high angular resolution and thus better isolation of detected emissions from celestial objects in order to construct the radio maps or images [20].

2.2 Brief History and Classification

The problem of DOA estimation has a century-long history and the first studies were related to radio direction finding (RDF). Initial efforts were done for RDF by Marconi (1906) and Bellini & Tosi (1909) using directional properties of some available antennas like dipoles and loops [2] and these systems were generally used for long wave signals. However, when it came to use for higher frequencies, some unknown effects (as learned later, multipath due to reflections from the ionosphere) caused problem for determining the location of emitter during the day. This led to the invention of Adcock antenna (four separate monopole antennas instead of two loops) and the corresponding DF system (1919) prevented sky wave paths via suppressing the horizontal components [21]. After that, a significant progress in RDF was suggested originally for locating lightning by Robert Watson-Watt by using the fifth antenna to eliminate the ambiguities [22]. The special Watson-Watt RDF systems named as HF/DF or huff-duff were then utilized for military roles in late 1930s. When more DF accuracy was required, pseudo-Doppler RDF systems were developed and they were widely used in place of huff-duff systems after WWII. A pseudo-Doppler system is based on a phase-based DF method in which a bearing estimation is realized by measuring the Doppler shift of the signal by rotating the antennas physically in early systems [23] or electrically switching between a set of antennas in modern systems. Following pseudo-Doppler systems which are actually single-channel interferometers, multi-channel interferometer systems were developed which consist of usually more than three omnidirectional antennas that are placed appropriately in the horizontal plane [24].

From the theoretical point of view, one of the first DOA methods is the conventional (Bartlett) beamformer and it is an old method going back to WWII and simply based on the Fourier analysis of the data sampled in the spatial domain [25]. Following this, to get a better DOA performance for closely spaced signal emitters, a well-known beamformer method called MVDR was proposed by J. Capon [26]. However, since its resolution capability still depends upon physical size of the antenna array (aperture) and SNR [6], the development of other DOA methods was driven. In the 1970s, Pisarenko proposed that DOA information can be extracted from second order statistics of the signal data [27] and with that, a new important class called subspace-based methods emerged including MUSIC, ESPRIT and their variants. Another common class of DOA methods (called as parametric methods) arisen from the weaknesses of the existing DOA methods and they were inspired from the maximum likelihood principles.

As a concluding point, there are various types of DOA methods proposed in the literature. They can be classified into two main categories, which are *spectral-based* and

parametric approaches [6] as shown in Figure 2.1.

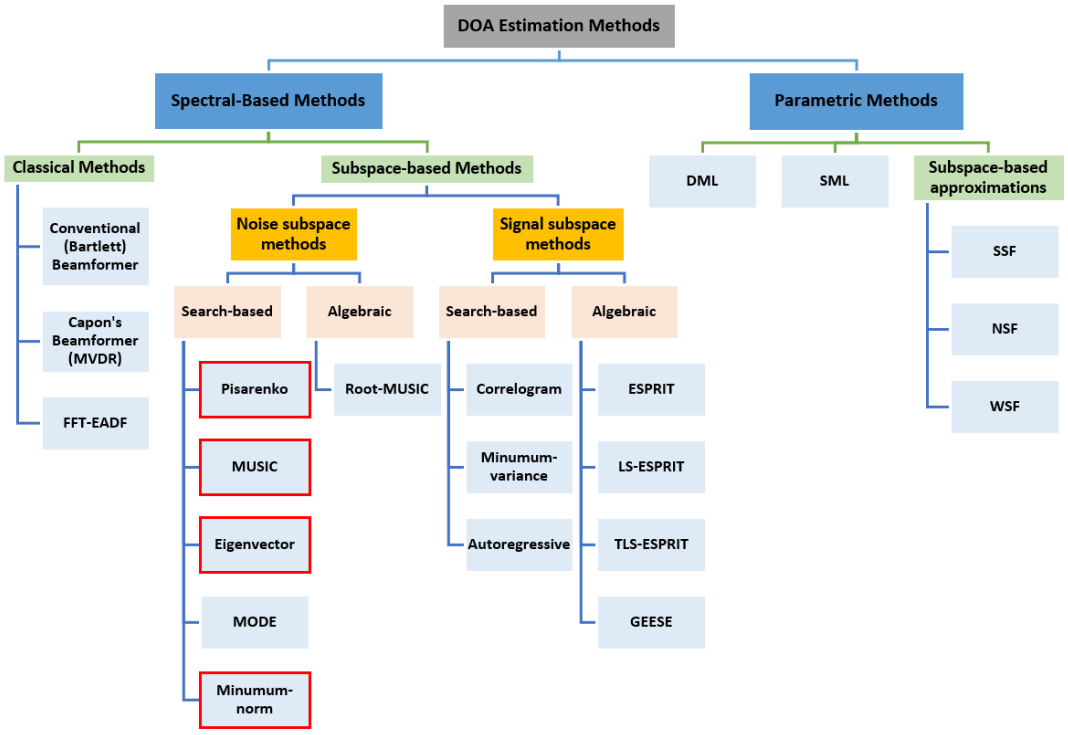


Figure 2.1: The classification of DOA methods.

The methods shown in red boxes are the ones to be considered in this thesis.

2.3 System Model

Sensor Array Signal Processing is an active research area in which some unknown signal parameters are extracted using measurement data obtained from an array of sensors placed in space [28] and these sensors are used for a sampling operation in space obeying the spatial version of the Nyquist Theorem. The direction of arrival (DOA) of the emitter signals is one of most well-known signal parameters studied in the literature. Estimating this parameter is generally accepted as an inverse problem and its solution needs a parametric model fed with the array output to obtain the parameter of interest [29]. Before diving into the data model and related assumptions, it is better to give some definitions first.

2.3.1 Definitions

- **Sensor:** A device which is capable of measuring a physical quantity (light, EM field, sound, pressure, etc.) and converting this into an electrical signal which is sampled and digitized later on. Some sensor examples are antennas (EM waves) , microphones (acoustic waves), hydrophones (ultrasonic waves), and

camera/photodiode (light) [30].

- **Sensor Array:** A collection of sensors placed in a certain type of array geometry to collect information which is not obtained using just a single sensor. The main purpose is to filter signals in both space and time.
- **Array Aperture:** The spatial region occupied by the array and it is defined in the continuous space domain whereas the sensor array is in the discrete space domain (at least in our context).
- **Array Geometry:** It is how the sensor array is configured in space. This can be linear (1D), planar (2D) or volumetric (3D).
- **Sensor Placement:** It defines how the space between the sensors is arranged. It can be uniform, non-uniform or random spacing.
- **Spherical Coordinate System:** It is the physical coordinate system and in sensor array signal processing, this system is commonly used. In this system, a point defined as $\mathbf{x} = (x, y, z)$ in Cartesian coordinate system can be represented in the spherical coordinate system as $\mathbf{r} = (r, \theta, \phi)$ where r is radial distance, ϕ is azimuth angle and θ is elevation/inclination angle, satisfying that $r \in [0, \infty)$, $\phi \in [0, 2\pi)$ and $\theta \in [0, \pi)$. Transformation between two coordinate systems can be realized by the equations in Equation 2.1 and it is represented visually in Figure 2.2.

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ y &= r \sin\theta \sin\phi \\ z &= r \cos\theta \end{aligned} \tag{2.1}$$

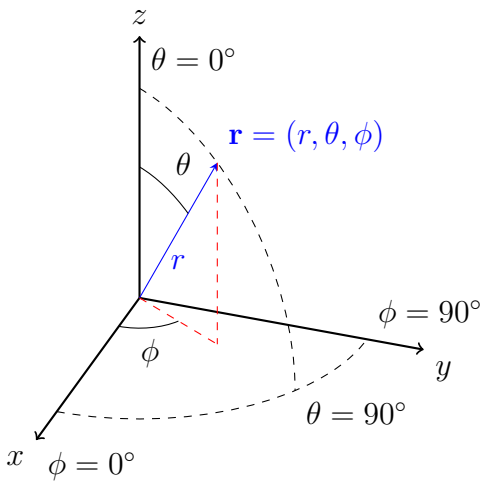


Figure 2.2: Spherical coordinate system (r, ϕ, θ) in relation with Cartesian coordinate system.

2.3.2 Assumptions

In order to construct a convenient theoretical model, some significant assumptions need to be made beforehand and once again, the data model and all algorithms to be mentioned in this thesis are investigated from the EM (electromagnetic) point of view.

1. **Narrowband Assumption:** The d incoming signals from d sources assumed to have the same f_c (carrier frequency) such that all the frequency contents are accumulated in the neighborhood of the same f_c and these sources in this way are also referred to as monochromatic. This condition can be achieved only if the amplitude and phase (information-carrying) components of the signals change slowly w.r.t. time spent for the impinging signals to travel from a sensor to another one [1].

2. **Far-Field Assumption:** The d signal sources are assumed to be far away from the sensor array such that their wavefronts can be modeled as planar (plane wave assumption) and by this way, arriving fields of the d signals can be considered as parallel to each other. Therefore, constant phase difference between consecutive elements can be assumed in the model. The far-field assumption is accepted as valid if the source is located at a distance larger than $2D^2/\lambda$ from the antenna array where D is the typical dimension (aperture) of the array and λ is the wavelength of the RF signal [31].

3. **Point Emitters:** RF sources in the model can be assumed to be point emitters, which means they are dimensionless and have an equal radiation in all directions. This assumption is acceptable in our case since the antennas of the radio sources are generally smaller than one wavelength.

4. **Number of Sources:** The number of sources is assumed to be known. If it was taken as unknown, a detection problem would be also involved and be solved by some methods like Akaike's Information Criterion [6] in addition to an estimation problem. This additional problem is out of the scope of this thesis.

5. **Linear and Isotropic Transmission Medium:** The transmission medium where the propagation from the sources to the antenna array occurs is assumed to be isotropic and linear. In other words, the physical properties of the medium do not vary in different directions and possible nonlinear effects are ignored such that the superposition of the waves at a certain point is possible [1].

6. **Uncorrelated sources:** The signals impinging on the antenna array are assumed to be uncorrelated (incoherent) such that there is no strong relation between them like being scaled and delayed version of each other. Even if this is practically not feasible since most of the realistic scenarios include some disturbing effects like multipath propagation, it is not a concern since the focus of this thesis stays more on the computation- and performance-side of the algorithms. Therefore, this assumption seems to be acceptable.

7. **Noise assumption:** Data collected from the array elements in the model includes both signal and noise components. The latter component in the model is

assumed to be an additive noise of a complex white Gaussian process (AWGN channel). This zero-mean noise is spatiotemporally stationary and uncorrelated with the signals. The noise components observed at the array elements are uncorrelated among them and they have a common variance σ_N^2 [1].

2.3.3 Data Model

Let us consider the scenario in Figure 2.3, where a passive antenna array with m elements placed in a random geometry and d point RF sources radiating from such a distance to the array that far-field assumption given in Section 2.3.2 holds.

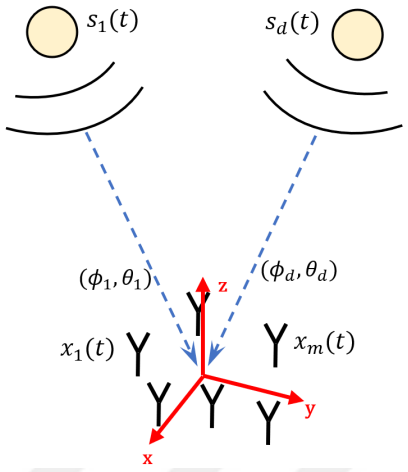


Figure 2.3: A passive antenna array receiving signals from point emitters.

Each antenna output in the given scenario is modeled as the LTI (linear time invariant) system response. Let us denote the impulse response of kth antenna to the incoming signal $s_i(t)$ as $h_{ki}(t)$ and propagation delay of the ith signal at kth antenna measured w.r.t. a certain reference point as τ_{ki}. Since an antenna element is considered as an LTI system, its output can then be expressed as a superposition (Equation 2.2).

$$x_k(t) = \sum_{i=1}^{d} h_{ki}(t) * s_i(t - \tau_{ki}) + n_k(t) \tag{2.2}$$

where (*) is used to represent the convolution operator and $n_k(t)$ is additive noise which is uncorrelated with $s_i(t)$.

It is also possible to put all antenna outputs in a compact matrix form as:

$$\boldsymbol{X}(t) = \begin{bmatrix} x_1(t) \\ . \\ . \\ . \\ x_m(t) \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{d} h_{1i} * s_i(t - \tau_{1i}) \\ \vdots \\ \sum_{i=1}^{d} h_{mi} * s_i(t - \tau_{mi}) \end{bmatrix} + \begin{bmatrix} n_1(t) \\ . \\ . \\ . \\ n_m(t) \end{bmatrix} \tag{2.3}$$

Narrowband assumption (Section 2.3.2) applied on the incoming signal $s(t)$ means that signal envelopes do not alter significantly while the wavefronts travel from one element to another and this allows the propagation delay of the signal to be modeled as only a phase shift of the carrier frequency [28]. Under the narrowband assumption, the adaptation of the model to complex signals makes us to reach the following parameterized data model for an array of m omnidirectional antennas at a specific time instant t:

$$\begin{aligned} \boldsymbol{X}(t) &= \begin{bmatrix} x_1(t) \\ . \\ . \\ . \\ x_m(t) \end{bmatrix} = \sum_{i=1}^{d} \begin{bmatrix} a_1(\phi_i, \theta_i) \\ . \\ . \\ . \\ a_m(\phi_i, \theta_i) \end{bmatrix} s_i(t) + \begin{bmatrix} n_1(t) \\ . \\ . \\ . \\ n_m(t) \end{bmatrix} \\ &= [\boldsymbol{a}(\phi_1, \theta_1) \ . \ . \ . \ \boldsymbol{a}(\phi_d, \theta_d)] [s_1(t) \ . \ . \ . \ s_d(t) \ ]^T + \boldsymbol{W}(t) \\ &= \boldsymbol{A}(\boldsymbol{\phi}, \boldsymbol{\theta}) \boldsymbol{S}(t) + \boldsymbol{W}(t) \end{aligned} \tag{2.4}$$

The derivation of the given parametric model is skipped for the sake of simplicity. All parameters given in Equation 2.4 are tabulated with their explanations and matrix expressions as in Table 2.1.

Table 2.1: Data model parameters with their explanations.

Parameters	**Explanations**	**Matrix Forms**
$\boldsymbol{X}(t)$ $(m\ x\ 1)$:	array output matrix	$[x_1(t)\ \ x_2(t) \ldots\ x_m(t)]^T$
$\boldsymbol{S}(t)$ $(d\ x\ 1)$:	incident complex signals	$[s_1(t)\ \ s_2(t) \ldots\ s_d(t)]^T$
$\boldsymbol{W}(t)$ $(m\ x\ 1)$:	AWG noise	$[n_1(t)\ \ n_2(t) \ldots\ n_m(t)]^T$
$(\boldsymbol{\phi}, \boldsymbol{\theta})$ $(d\ x\ 1)$:	unknown DOAs	$[(\phi_1,\ \theta_1)\ ,\ \ldots,\ (\phi_d,\ \theta_d)]^T$
$\boldsymbol{A}(\boldsymbol{\phi}, \boldsymbol{\theta})$ $(m\ x\ d)$:	array manifold matrix	$[\boldsymbol{a}(\phi_1,\ \theta_1), \ldots\ \boldsymbol{a}(\phi_d,\ \theta_d)]$

$$\boldsymbol{a}(\phi_i, \theta_i) = \begin{bmatrix} \exp\left\{ j\frac{2\pi}{\lambda}\ (x_1 \sin\theta_i \sin\phi_i + \ y_1 \cos\theta_i \sin\phi_i + z_1\ cos\ \phi_i) \right\} \\ \vdots \\ \exp\left\{ j\frac{2\pi}{\lambda}\ (x_m \sin\theta_i \sin\phi_i + \ y_m \cos\theta_i \sin\phi_i + z_m\ cos\ \phi_i) \right\} \end{bmatrix} \tag{2.5}$$

In Table 2.1, the array manifold matrix $\boldsymbol{A}(\boldsymbol{\phi}, \boldsymbol{\theta})$ consists of d complex array steering vectors stacked in the columns of the matrix. Each vector $\boldsymbol{a}(\phi_i, \theta_i)$ is a representation for an array response (considering m elements) to the ith source coming at the angle (ϕ_i, θ_i) and hence the steering vectors carry information about complex directional characteristics of the array elements in relative manner [2].

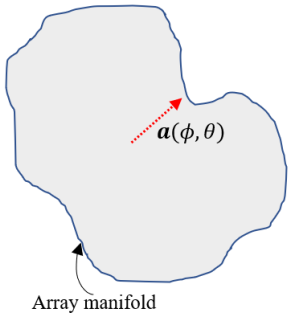


Figure 2.4: Visual representation of an array manifold.

Array steering vector $\boldsymbol{a}(\phi_i, \theta_i)$ for the unit amplitude signal can be visually represented as in Figure 2.4 and it can be expressed in a general manner (for all geometries) as in Equation 2.5 where λ is the wavelength of the incoming signal(s) and $\{x_k,\ y_k, z_k\}$ for all $k \in \{1, 2, ..., m\}$ are the coordinates of the array elements.

In a more realistic scenario, it is required to observe the array output $\boldsymbol{X}(t)$ for a duration to collect N samples at $t_1, ..., t_N$. Each vector observation is known as a *snapshot* of the array output [28]. Therefore, it is better to describe Equation 2.4 in a discrete form as:

$$\boldsymbol{X}_N = [\boldsymbol{X}(t_1) \ ... \ \boldsymbol{X}(t_N)] = \boldsymbol{A}(\boldsymbol{\phi}, \boldsymbol{\theta})\boldsymbol{S}_N + \boldsymbol{W}_N \tag{2.6}$$

The process described above can be clearly seen as a multi-channel random process, whose characteristics are determined from first and second order statistics of the underlying signal and noise components.

2.4 Noise Subspace-based DOA Methods

In the single emitter (source) case, the classical and subspace-based (super-resolution) methods given in Figure 2.1 have a similar estimation performance. On the other hand, the subspace-based methods have considerably better performance compared to the classical methods in the multiple emitter case, especially when the emitters are angularly close to each other. This performance difference comes from the fact that subspace-based methods can have a resolution performance[1] which goes beyond the Rayleigh bound[2], that is why these methods are referred to as "super-resolution"

[1] It is usually measured in terms of the smallest distinguishable angle.

[2] Rayleigh resolution bound is the term in wave physics for the discernibility limit of two point light sources and in our case, it corresponds to approximately one beamwidth [2].

methods [32]. The performance of these methods is generally limited by how accurately the signal and noise subspaces are separated in a noisy scenario [33]. As visually represented in Figure 2.1, subspace (super-resolution) methods are divided into two groups, namely noise-subspace and signal-subspace methods. Within the scope of this thesis, search-based four noise subspace methods will be studied and they are given as:

- Pisarenko Harmonic Decomposition (**PHD**)
- MUltiple SIgnal Classification (**MUSIC**)
- EigenVector (**EV**)
- Minimum-Norm (**MN**)

All the calculations to be mentioned in these algorithms are based upon time samples of the arriving signals because DOA values for the incoming signals may differ with time. When the received signals at the array elements change, the corresponding DOA estimation may change as well. Therefore, there is a need to calculate the array spatial covariance matrix $\boldsymbol{R_{XX}}$ and this is expressed as given below.

$$\begin{aligned} \boldsymbol{R_{XX}} &= E\ \left[\boldsymbol{X}\left(t\right)\ \boldsymbol{X}^{H}\left(t\right)\right] = \boldsymbol{A}\left(\phi,\theta\right)\boldsymbol{R_{SS}}\boldsymbol{A}^{H}\left(\phi,\theta\right) + \boldsymbol{R_{WW}} \\ \boldsymbol{R_{SS}} &= E\ \left[\boldsymbol{S}\left(t\right)\ \boldsymbol{S}^{H}\left(t\right)\right]\ \ \&\ \ \boldsymbol{R_{WW}} = \sigma_N^2\,\boldsymbol{I_m} \end{aligned} \tag{2.7}$$

where $(.)^H$ denotes the Hermitian transpose (or conjugate transpose), σ_N^2 denotes noise variance and $\boldsymbol{I_m}$ is m-by-m identity matrix. It is assumed that there exists no correlation between the signal sources and therefore, the signal covariance matrix $\boldsymbol{R_{SS}}$ is a full-rank (non-singular) matrix. The noise covariance matrix $\boldsymbol{R_{WW}}$ is defined as a diagonal matrix whose all elements are σ_N^2 by the spatially white noise assumption. Practically, the actual covariance matrix is **not available** and it is necessary to **estimate** the sample covariance matrix $\widehat{\boldsymbol{R}}_{\boldsymbol{XX}}$ (or simply $\widehat{\boldsymbol{R}}$) from the received data over $\boldsymbol{N}$ **samples** as given below:

$$\boldsymbol{R_{XX}}\ \approx\ \widehat{\boldsymbol{R}}_{\boldsymbol{XX}} = \widehat{\boldsymbol{R}} = \frac{1}{N}\sum_{k=1}^{N}\boldsymbol{X}\left(t\right)\ \boldsymbol{X}^{H}\left(t\right) \tag{2.8}$$

Again to remember, our data model adopts a scenario in which there are m antenna elements receiving a collection of plane wavefronts radiated by $d\,(d < m)$ sources located in the far-field region of the antenna array. The Hermitian characteristics of the (estimated) sample covariance matrix $\widehat{\boldsymbol{R}}$ allows us to apply the eigendecomposition and with this, it could be decomposed as:

$$\widehat{\boldsymbol{R}} = \ \widehat{\boldsymbol{U}}\widehat{\boldsymbol{\Lambda}}\widehat{\boldsymbol{U}}^{H} \tag{2.9}$$

where $\widehat{\boldsymbol{U}}$ is an m-by-m matrix having eigenvectors of $\widehat{\boldsymbol{R}}$ and $\widehat{\boldsymbol{\Lambda}}$ is an m-by-m diagonal matrix with diagonal elements $\{\lambda_1, \lambda_2, ..., \lambda_m\}$ as the eigenvalues of $\widehat{\boldsymbol{R}}$. The

eigenvalues can be sorted in a descending way such that $\lambda_{max} \geq ... \geq \lambda_{min} > 0$. By the noise assumption described in Section 2.3.2, the noise term in the model is uncorrelated with the signals. Therefore, the sample covariance matrix can be expressed as follows [6].

$$\widehat{\boldsymbol{R}} = \widehat{\boldsymbol{U}}_s \widehat{\boldsymbol{\Lambda}}_s \widehat{\boldsymbol{U}}_s^H + \widehat{\boldsymbol{U}}_n \widehat{\boldsymbol{\Lambda}}_n \widehat{\boldsymbol{U}}_n^H \tag{2.10}$$

The first term in Equation 2.10 represents the signal subspace which is spanned by the d eigenvectors corresponding to the d largest eigenvalues of $\widehat{\boldsymbol{R}}$. These d eigenvalues can be expressed as $\lambda_i = {\lambda_i}^s + \sigma_N^2$ where ${\lambda_i}^s$ are the eigenvalues of $\boldsymbol{R_{SS}}$ defined in Equation 2.7. The second term, on the other hand, represents the noise subspace which is spanned by the remaining $m - d$ eigenvectors corresponding to the remaining smallest $m - d$ eigenvalues of $\widehat{\boldsymbol{R}}$. These $m - d$ eigenvalues are equal to σ_N^2. This eigendecomposition approach presented here is used for all the noise subspace methods in our consideration.

2.4.1 Pisarenko Harmonic Decomposition (PHD) Method

The PHD method is known as the first subspace-based method that was developed in the DOA history and it was proposed by V. Pisarenko in 1973 for the retrieval of harmonics from a covariance function based on the Caratheodory Theorem used in the trigonometric moment problem [27].

Within this method, it is assumed that the received signal $\boldsymbol{X}(t)$ is composed of $\boldsymbol{d}$ complex exponentials with an existing white Gaussian noise and the array is composed of m elements. Theoretically, PHD method mainly uses the eigenstructure of the m-by-m sample covariance matrix $\widehat{\boldsymbol{R}}$. The main historical purpose of this method is to estimate the frequencies of the sinusoidal signals in damped and underdamped cases when the spatial covariance matrix is provided [6].

After the eigenvalue decomposition (EVD) operation on $\widehat{\boldsymbol{R}}$ and sorting the resulting eigenvalues in decreasing order, as the distinct property of the PHD method, the eigenvector ($\boldsymbol{e_{min}}$) associated with the smallest eigenvalue (λ_{min}) is taken into the consideration. If the smallest one is repeated, any unit-norm vector constituted as a linear combination of the corresponding eigenvectors can be used as $\boldsymbol{e_{min}}$ [32].

The PHD method assumes that the dimension of the noise subspace is equal to "one" and it is spanned by the eigenvector $\boldsymbol{e_{min}}$ corresponding to the minimum eigenvalue λ_{min} regardless of the number of incoming signals under our consideration (i.e. assume $m = d + 1$). This specific eigenvector $\boldsymbol{e_{min}}$ is necessarily orthogonal to all of the signal subspace eigenvectors [34]. Using this, the ***pseudospectrum*** or ***eigenspectrum*** of the PHD method can be expressed as:

$$P_{PHD}(\boldsymbol{\theta}) = \frac{1}{\boldsymbol{a}(\boldsymbol{\theta})^H \boldsymbol{N}\, \boldsymbol{a}(\boldsymbol{\theta})} = \frac{1}{\boldsymbol{a}(\boldsymbol{\theta})^H (\boldsymbol{e_{min}} \boldsymbol{e_{min}^H})\, \boldsymbol{a}(\boldsymbol{\theta})} = \frac{1}{\left\| \boldsymbol{a}(\boldsymbol{\theta})^H \boldsymbol{e_{min}} \right\|^2} \tag{2.11}$$

where $\boldsymbol{a}(\boldsymbol{\theta})$ is reduced form of the array steering vector $\boldsymbol{a}(\phi_i, \theta_i)$ and $\boldsymbol{N}$ represents the generic matrix formed by the noise eigenvectors, which is generally used in search-based noise subspace methods.

In the single emitter case, the denominator of $P_{PHD}(\boldsymbol{\theta})$ takes a minimum value (ideally zero) when the orthogonality of a steering vector and $\boldsymbol{e_{min}}$ is nearly achieved and a sharp peak is ideally observed in the pseudospectrum. In the multiple emitter case, the smallest multiple values for the denominator are considered and hence, multiple peaks are observed in the pseudospectrum. These peaks correspond to nothing but the DOA values of the emitters.

The resulting technique has a relatively limited usefulness since it shows an apparent sensitivity to noise. However, from the theoretical point of view, it has provided valuable insights into the the DOA estimation problem [34].

2.4.2 MUltiple SIgnal Classification (MUSIC) Method

The MUSIC method was first proposed by Schmidt in 1979 [35] and it is nothing but an improved version of the PHD method. Since that time, it is known as one of the most popular DOA estimation algorithms. The idea behind is similar to that of the PHD method.

Let us remember that the sample covariance matrix can be spectrally decomposed as $\widehat{\boldsymbol{R}} = \widehat{\boldsymbol{U}} \widehat{\boldsymbol{\Lambda}} \widehat{\boldsymbol{U}}^H$ and it is possible to partition the matrix of eigenvectors $\widehat{\boldsymbol{U}}$ as

$$\widehat{\boldsymbol{U}} = \left[\widehat{\boldsymbol{U}}_{\boldsymbol{s}}, \widehat{\boldsymbol{U}}_{\boldsymbol{n}} \right] \tag{2.12}$$

where $\widehat{\boldsymbol{U}}_{\boldsymbol{s}}$ is the m x d matrix composed of the eigenvectors corresponding to d largest eigenvalues and $\widehat{\boldsymbol{U}}_{\boldsymbol{n}}$ is the m x $(m-d)$ matrix composed of the eigenvectors corresponding to $m-d$ smallest eigenvalues. When the sources are incoherent, it is important to note that $\widehat{\boldsymbol{U}}_{\boldsymbol{s}}$ and the array manifold A share the same range space [6] given as:

$$\mathcal{R}\{\widehat{\boldsymbol{U}}_{\boldsymbol{s}}\} = [\mathbf{a}(\phi_1, \theta_1), ..., \mathbf{a}(\phi_d, \theta_d)] \tag{2.13}$$

It is possible to refer $\widehat{\boldsymbol{U}}_{\boldsymbol{s}}$ as the signal subspace and $\widehat{\boldsymbol{U}}_{\boldsymbol{n}}$ as the noise subspace.

Since $\boldsymbol{U}$ is known to be a unitary matrix, then these subspaces are then orthogonal to each other, so that

$$\mathbf{a}^H(\phi_i, \theta_i)\, \boldsymbol{U_n} = 0 \;\text{ for }\; i = 1, ..., d \tag{2.14}$$

Being different from the PHD method, instead of using a single eigenvector, the MUSIC method uses all noise eigenvectors to construct the matrix $\boldsymbol{N}$. Keeping this in mind, the pseudospectrum expression (also called as the MUSIC spectrum) can be derived as:

$$\begin{aligned} P_{MUSIC}(\boldsymbol{\theta}) &= \frac{1}{\boldsymbol{a}(\boldsymbol{\theta})^H\, \boldsymbol{N}\, \boldsymbol{a}(\boldsymbol{\theta})} \\ &= \frac{1}{\boldsymbol{a}(\boldsymbol{\theta})^H\, (\, \sum_{i=d+1}^{m} \boldsymbol{e_i} \boldsymbol{e_i}^H)\, \boldsymbol{a}(\boldsymbol{\theta})} = \frac{1}{\boldsymbol{a}(\boldsymbol{\theta})^H\, \boldsymbol{E_n}\, \boldsymbol{E_n}^H\, \boldsymbol{a}(\boldsymbol{\theta})} \end{aligned} \tag{2.15}$$

where $\boldsymbol{a}(\boldsymbol{\theta})$ represents $\boldsymbol{a}(\phi_i, \theta_i)$ and assuming that the noise eigenvectors are obtained by the EVD of the sample covariance matrix as $\{\boldsymbol{e_{d+1}}, \boldsymbol{e_{d+2}}, \ldots, \boldsymbol{e_m}\}$, the matrix $\boldsymbol{E_n}$ is the combination of the noise eigenvectors ($\boldsymbol{e_i}$) in the columns as:

$$\boldsymbol{E_n} = [\, \boldsymbol{e_{d+1}} \vdots\, \boldsymbol{e_{d+2}} \vdots\, \ldots \vdots\, \boldsymbol{e_m}] \tag{2.16}$$

Moreover, the matrix $\boldsymbol{N}$ in Equation 2.15 is calculated by a summation term which is actually the orthogonal projection onto the range space of the $\boldsymbol{E_n}$. Similar to the PHD case, when the orthogonality with the array steering vector is nearly satisfied, the denominator term drops to a small value and the pseudospectrum shows a peak for that. These peaks are the DOA results of the MUSIC method.

The MUSIC method generally shows better performance than the PHD method, and it has an estimation performance which approaches to the optimum solution asymptotically [35]. The disadvantage of the noise subspace-based methods including MUSIC is the requirement for sufficient number of samples (snapshots) since their computations strongly depend upon the estimated (sample) covariance matrix of the received data [2].

2.4.3 EigenVector (EV) Method

In addition to PHD and MUSIC methods, some other eigenvector methods were proposed for the frequency estimation of complex exponentials in the noise. One of these methods is the EigenVector (EV) method [36]. This method is quite similar to the MUSIC algorithm.

In contrast to the MUSIC method, the EV method makes use of inverse noise eigenvalues obtained by the EVD of the sample covariance matrix $\widehat{\boldsymbol{R}}$. These values are basically utilized for weighting the corresponding eigenvectors. In the MUSIC method,

these weights are considered as unity. The expression for calculating EV-spectrum is given as:

$$P_{EV}(\boldsymbol{\theta}) = \frac{1}{\boldsymbol{a}(\boldsymbol{\theta})^H \, \boldsymbol{N} \, \boldsymbol{a}(\boldsymbol{\theta})} = \frac{1}{\boldsymbol{a}(\boldsymbol{\theta})^H \, \left(\sum_{i=d+1}^{m} \frac{1}{\lambda_i} \boldsymbol{e_i} \boldsymbol{e_i}^H \right) \boldsymbol{a}(\boldsymbol{\theta})} = \frac{1}{\boldsymbol{a}(\boldsymbol{\theta})^H \, \boldsymbol{E_n'} \, \boldsymbol{E_n}^H \, \boldsymbol{a}(\boldsymbol{\theta})} \tag{2.17}$$

where $\boldsymbol{a}(\boldsymbol{\theta})$ represents $\boldsymbol{a}(\phi_i, \theta_i)$ and the matrix $\boldsymbol{N}$ in this method is formed using $\boldsymbol{E_n'}$ instead of $\boldsymbol{E_n}$. In fact, this is the main difference from the MUSIC spectrum expression and it is realized by just weighting each noise eigenvector $\boldsymbol{e_i}$ with the inverse of the corresponding eigenvalue as given in Equation 2.18. Note that $\boldsymbol{E_n}^H$ matrix is computed by taking the Hermitian transpose of $\boldsymbol{E_n}$.

$$\boldsymbol{E_n'} = \left[(1/\lambda_{d+1}) \, \boldsymbol{e_{d+1}} \vdots \, (1/\lambda_{d+2}) \, \boldsymbol{e_{d+2}} \vdots \, \ldots \vdots \, 1/\lambda_m \, \boldsymbol{e_m} \right] \tag{2.18}$$

By the weighting coefficients (inverse eigenvalues) inside the EV-spectrum computation, the method claims to have fewer spurious peaks than MUSIC and the method gives a better shape to the noise spectrum by less pronouncing the low-level source and physical noise effects [36].

2.4.4 Minimum-Norm (MN) Method

Another noise subspace-based method studied within the scope of this thesis is the Minimum-Norm (MN) method. This method was firstly proposed by Kumerasan and Tufts [37] in order to suggest a solution to the frequency estimation problem.

In this algorithm, rather than constructing an eigenspectrum using all eigenvectors spanning the noise subspace as in the MUSIC & EV methods, the MN method makes use of a single vector $\boldsymbol{v}$ which is constrained to lie within the noise-subspace. The DOA estimation by this method is realized by determining where the peaks are obtained in the MN-eigenspectrum and this spectrum is computed as follows.

$$P_{MN}(\boldsymbol{\theta}) = \frac{1}{\boldsymbol{a}(\boldsymbol{\theta})^H \, \boldsymbol{N} \, \boldsymbol{a}(\boldsymbol{\theta})} = \frac{1}{\boldsymbol{a}(\boldsymbol{\theta})^H \, \boldsymbol{v} \, \boldsymbol{v}^H \, \boldsymbol{a}(\boldsymbol{\theta})} = \frac{1}{\left\| \boldsymbol{a}(\boldsymbol{\theta})^H \, \boldsymbol{v} \right\|^2} \tag{2.19}$$

where $\boldsymbol{a}(\boldsymbol{\theta})$ represents $\boldsymbol{a}(\phi_i, \theta_i)$, the matrix $\boldsymbol{N}$ in this method is formed using the vector $\boldsymbol{v}$ and the problem here is to determine which vector $\boldsymbol{v}$ in the noise subspace minimizes the spurious effects on the spectrum peaks. The approach that is followed for this minimization is to find the vector $\boldsymbol{v}$ that fulfills the constraints summarized below [34]:

1. The vector $\boldsymbol{v}$ must lie in the "noise subspace", hence it is orthogonal to the projection onto the signal subspace spanned by array steering vectors.

2. The vector $\boldsymbol{v}$ has "minimum norm".

3. The first element of this vector should be "unity".

The first constraint guarantees that "d roots" of $\boldsymbol{V}(z)$ (z-transform of the coefficients in vector $\boldsymbol{v}$) lie *on* the unit circle. The second constraint ensures that the "spurious roots" of $\boldsymbol{V}(z)$ lie *inside* the unit circle. Lastly, the third constraint guarantees that the minimum-norm solution is the "non-zero vector".

The constrained minimization problem given above can be solved starting from the first constraint that can be expressed as $\boldsymbol{v} = \boldsymbol{P_n}\, \boldsymbol{w}$ where $\boldsymbol{P_n} = \boldsymbol{E_n}\, \boldsymbol{E_n^H}$ is the matrix projecting $\boldsymbol{w}$ onto noise subspace and the matrix $\boldsymbol{E_n}$ is constructed by inserting each noise eigenvector $\boldsymbol{e_i}$ of the sample covariance matrix into the columns of $\boldsymbol{E_n}$. After a few additional steps (see [34] for detail), the minimization problem is solved as:

$$\boldsymbol{w} = \lambda\, \boldsymbol{u_1} \tag{2.20}$$

where $\boldsymbol{u_1} = [1,\, 0,\, \ldots\, ,\, 0]^T$, which is a special vector for emphasizing the first element of the vector $\boldsymbol{v}$ and λ is weight vector calculated by $\lambda = (\boldsymbol{u_1^H}\, \boldsymbol{P_n}\, \boldsymbol{u_1})^{-1}$. Finally, the pseudospectrum expression for the MN method is obtained as follows.

$$\begin{aligned} \boldsymbol{v} = \boldsymbol{P_n}\, \boldsymbol{w} = \lambda\, \boldsymbol{P_n}\, \boldsymbol{u_1} = \frac{\boldsymbol{P_n}\, \boldsymbol{u_1}}{\boldsymbol{u_1^H}\, \boldsymbol{P_n}\, \boldsymbol{u_1}} \quad \overset{yields}{\longrightarrow} \\ P_{MN}\,(\boldsymbol{\theta}) = \frac{1}{\left\| \boldsymbol{a}(\boldsymbol{\theta})^H\, \boldsymbol{v} \right\|^2} = \frac{1}{\left\| \boldsymbol{a}(\boldsymbol{\theta})^H\, \boldsymbol{E_n}\, \boldsymbol{E_n^H}\, \boldsymbol{u_1} \right\|^2} \end{aligned} \tag{2.21}$$

The expression given above is nothing but projection of the unit vector onto the noise subspace, normalized in order to satisfy that the first coefficient is equal to "unity".

This method is generally recommended if the configuration is Uniform Linear Array (ULA). For two closely spaced equal-power signals, the MN method claims to succeed a lower SNR threshold, which is the metric for minimum SNR-level required to separate two sources [38].

CHAPTER 3

OPENMP AND CUDA

In order to accelerate the bottleneck sections of the aforementioned algorithms on GPU, CUDA platform has been adopted (see Section 4.4). To enable a fair comparison, baseline C/C++ codes have also been implemented in both serial version and parallel version (using OpenMP).

In this chapter, some background information is given related to OpenMP and CUDA in order to provide a basis for the parallelization concepts to be mentioned in Chapter 4.

3.1 OpenMP

OpenMP (Open Multi-Processing) is an API initially developed for realizing parallel applications in the systems having multiprocessors with shared-memory support. It is actually not a programming language, but it can rather be seen as an extension to some programming languages, namely FORTRAN, C, and C++ [39]. Its specifications are maintained by a non-profit organization named as OpenMP Architecture Review Board (ARB) including some permanent and auxiliary members, some of which are AMD, IBM, Intel, NVIDIA, EPCC, RWTH Aachen and NASA [40].

OpenMP is an easy-to-use and mature API which requires only a limited amount of programming effort for parallelization. Most of the troublesome works required for the parallel programming are handled by the compiler [39]. Another advantage of OpenMP is that it enables the programmers to incrementally create the parallel programs from their serial code without forcing them to apply the parallelization for all parts at the same time. In most of the cases, the parallelization performance obtained by OpenMP satisfies its users and the scalability is achieved without too much effort [40].

All fundamental information related to OpenMP is given in the following.

3.1.1 Definitions

The following definitions are given from the the OpenMP perspective [40].

- **Thread:** An entity for execution having stack and *threadprivate memory* and it

actually corresponds to the smallest processing unit which might be scheduled by an OS.

- **Team:** A collection of thread(s) which participates in a parallel region execution.
- **Directive:** A base language (FORTRAN, C or C++) mechanism determining the behavior of a program.
- **Construct:** An executable directive and its related statements.
- **Structured Block:** An executable statement or compound in C/C++ that is starting with a single entry and ending with a single exit.
- **Region:** A code part during a specific time at which the execution of a given structured block sequence or a library routine encounters.
- **Memory:** A resource for storage in order to keep/write and read variables used by OpenMP threads.
- **Barrier:** A point determined in a program execution that a team encounters and beyond that point, no thread in a team can proceed until all ones in the team reach to the barrier.
- **Task:** A specific executable code instance and the corresponding data environment which are schedulable for execution by the threads.
- **Private Variable:** A variable providing a different storage block access for each task within the common parallel region.
- **Shared Variable:** A variable providing the same storage block access for each task within the common parallel region.
- **Threadprivate Variable:** A variable which is copied for each thread and it provides each with an access to a different storage block.

3.1.2 The OpenMP Programming Model

OpenMP is based upon the usage of multiple threads in the programming structure involving a shared memory system and it is not an automatic model such that it allows the programmer to have complete control over the parallelization. The OpenMP API with which the programmer directly interacts is mainly composed of the following three elements:

- **Compiler Directives:** They are mainly used to inform the compiler which instructions are requested for a parallel execution and how they are distributed among the principal threads [39]. Some of these directives are parallel region, work-sharing constructs, tasking, data-sharing attributes and so on. These compiler directives are used in the following syntax:

```
#pragma omp directive [clause]
```

where "pragma" is used to inform a compiler or preprocessor about some information on the processing to be governed by the directive. It does not express any logic and it is similar to `#include` and `#define`.

- **Runtime Routines:** The runtime routines are functions which can be used for querying some settings and for setting some runtime values by overriding the default or prespecified values [41]. Some example usages of these functions are number of threads, thread ID, nested parallelism, schedule, thread limit, dynamic thread adjustment and locking [40].

- **Environment Variables:** There are certain number of environment variables available to make some runtime settings to be initialized. If they are not set, their implementation-dependent default settings are used except the nested parallelism, and they have precedence over the run time function with the same functionality [41]. Some functional examples of these variables are thread number, type of scheduling, nested parallelism, thread limit, stacksize and dynamic adjustment for threads [40].

3.1.3 The OpenMP Execution Model

This model defines how an OpenMP program is executed. The OpenMP API is known to use the fork-join parallel execution model and multiple execution threads perform the implicitly- or explicitly-defined tasks. An important point to note is that a parallel program using OpenMP may produce different numeric results when executed with different number of threads due to possible associational variations in the numeric operations [40].

The fork-join execution model of OpenMP is illustrated in Figure 3.1. Here, an OpenMP program starts with a single master thread and it is initially executed in a sequential way. When it encounters with a parallel region, worker threads are created in the requested number and they hence form a team. OpenMP has SPMD (Single Program, Multiple Data) program model for its parallel regions and according to this, the same code is executed by all threads within these regions. This model is also commonly used for parallelization patterns in platforms like CUDA and MPI.

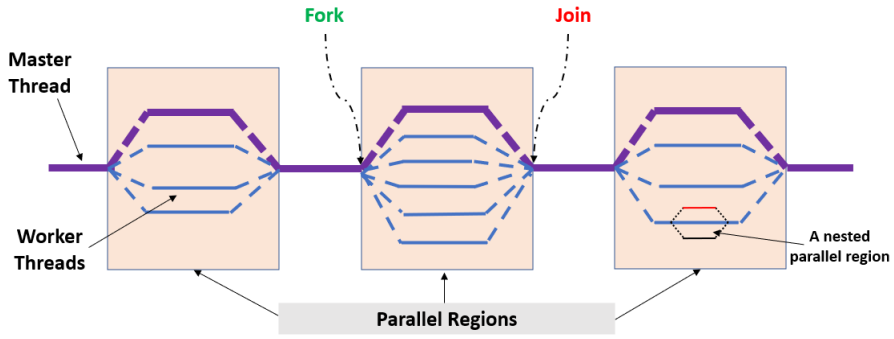


Figure 3.1: The OpenMP execution model.

3.1.4 The OpenMP Memory Model

OpenMP has the memory model as illustrated in Figure 3.2. As can be seen, there are two main types of memory: *shared* and *private*. Each thread (shown as T) has its own private memory and it can read and write (R/W) to that memory. However, there is no way to access the variables stored in the private memory by another thread. All threads have R/W access to the variables stored in shared memory. The memory type to be used for a specific variable depends upon either default rules or some clause explicitly defined on a construct [41].

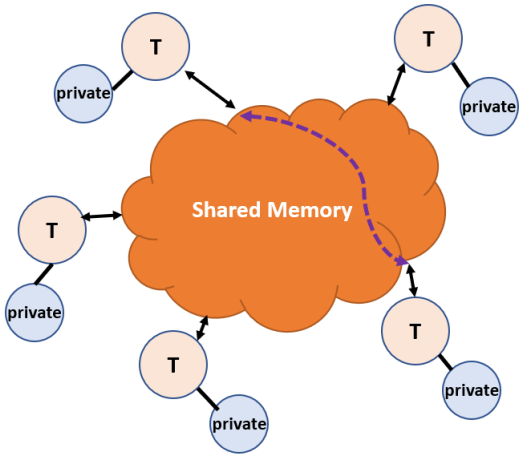


Figure 3.2: The OpenMP memory model.

3.2 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model introduced by NVIDIA© in 2007 for general purpose computing on the supported graphics processor units (GPUs). CUDA is supported for different popular languages (e.g., C/C++, FORTRAN, and Python) [42] and the developers can express the parallelism in their codes using some keywords that can be considered as extensions to their language. In this thesis, the C/C++ support version of the CUDA is considered.

The GPUs are inherently designed for high-speed graphical computations which are done in parallel. With the introduction of the CUDA platform, without any need for knowing graphics primitives, an opportunity for actively using these powerful graphics co-processors by everyday developers has come in sight [43]. With the advantage of a massive data parallelism provided by CUDA, it has been adopted in various areas where high computing performance is critical. Some examples for these areas are computational finance, deep/machine learning, climate modeling and supercomputing. This platform is composed of architecture (hardware design and structure) and programming model (how to develop software suitably for the hardware). A typical CUDA application developed by using this integrated model works upon two major

components, which are host and device. The host side consists of CPU and its associated memory units (RAM, cache, etc.) whereas the device side consists of GPU and its associated memory units (GDRAM, cache, etc.).

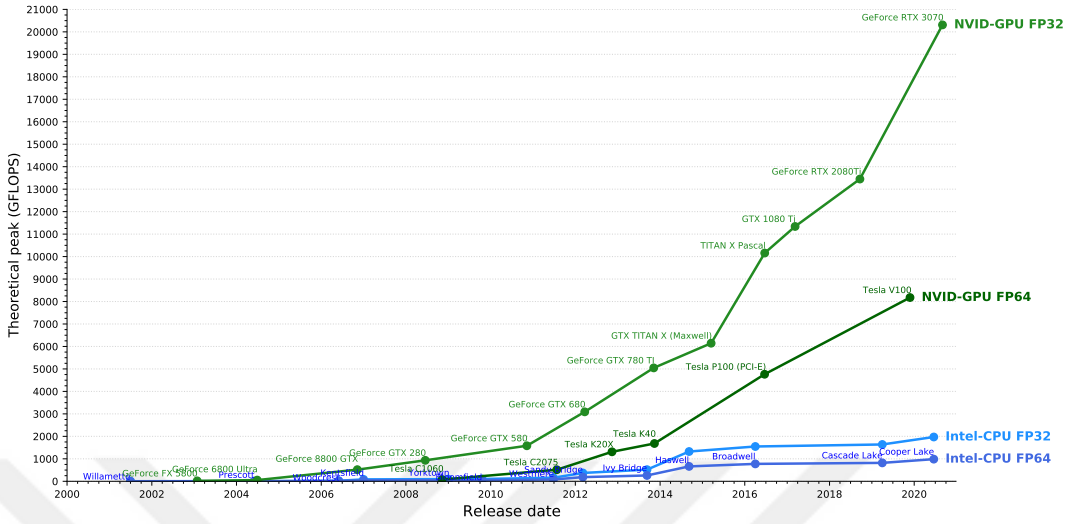


Figure 3.3: The theoretical peak performance history of CPUs and GPUs.

The plot given in Figure 3.3 is generated by using the data provided in [44] and the code in [45]. In this plot, the theoretical peak performance (GFLOPS) evolution of CPUs and GPUs is shown. It is seen that FP32 (single-precision) and FP64 (double-precision) peak performance gaps between Intel CPUs and NVIDIA GPUs have started to increase since 2010s. Today, a high-end GPU card of NVIDIA has about 10 times (FP32) and 8 times (FP64) higher theoretical peak performance in FLOPS compared to a high-end CPU of Intel.

There are several differences between the CPU and the GPU. Two processors are compared in Table 3.1.

Table 3.1: The comparison of the CPU and the GPU.

CPU	GPU
higher clock rate	moderate clock rate
computing a task rapidly	computing (many) tasks in mid-speed
lower latency with big caches	hidable latency with multi-threading
reasonable throughput	high throughput
few cores (today < 150)	many cores (today < 6000)
ordered consecutive tasks	unordered indepedent tasks

A typical simplified workflow for a CUDA application in heterogeneous mode is illustrated as follows.

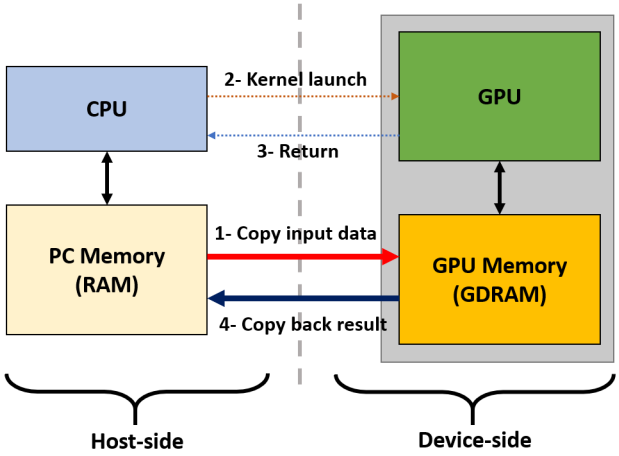


Figure 3.4: A typical workflow (1 to 4) for a CUDA application.

As can be seen in Figure 3.4, there are four basic steps in the workflow. Firstly, the input data on the host RAM to be used for the computation is copied to the pre-allocated memory space on the GPU memory. Secondly, the kernel(s) defined in the program is/are launched for the desired computations by the CPU. Thirdly, when the computations on the GPU are completed, a return is requested by the main program on the host. Lastly, this request is replied by the GPU and it copies the resultant data back to the host-side. In this integrated approach, fully serial or modestly-parallel parts in the overall code are executed on the host side and parts which are suitable for a much higher parallelization are executed using SPMD (Single Program Multiple Data) kernels on the device side. The data-parallel code portion is run on preferably thousands of CUDA threads to harness the power of the GPU.

All the details on CUDA are explained from different perspectives in the following four subsections.

3.2.1 CUDA Memory Model

Several types of memory are supported by CUDA and they are generally served to different needs in the GPU programming scheme to achieve high execution speed performance in the kernel(s). The overview of the CUDA device memory model [46] is visually given as follows.
In Figure 3.5, most of the memory types residing in the device-side are shown in relation to thread and block concepts. Some properties of these memories and two other ones (local and texture memory) are given in Table 3.2. In this table, off-chip memory is used to express the memory residing on the GDRAM, R/W is used for readable/writable and N/A means not applicable.

The R/W data communication between host and device is realized by a PCI-express interface, which plays a critical role in applications where an intensive amount of data needs to be copied into the GPU memory.

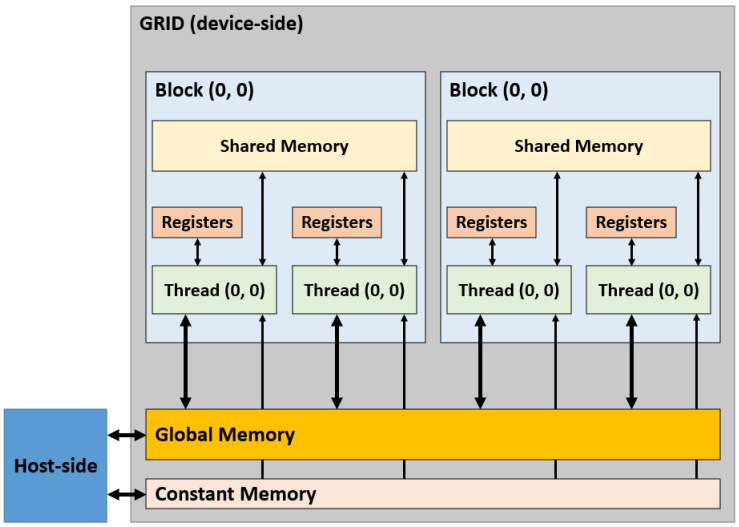


Figure 3.5: CUDA device memory model.

Table 3.2: CUDA memory types and their properties [47].

Memory	Location	Cached	Access	Latency	Scope	Lifetime
Register	On-chip	N/A	R/W	1 cycle	Thread	Thread
Local	Off-chip	Yes	R/W	100s	Thread	Thread
Shared	On-chip	N/A	R/W	1 cycle	Block	Block
Global	Off-chip	Yes	R/W	100s	All + host	Applic.
Constant	Off-chip	Yes	R	100s	All + host	Applic.
Texture	Off-chip	Yes	R	100s	All + host	Applic.

Due to their practical importance, some of the GPU memory types in Table 3.2 are explained with some details as follows [42].

- **Local Memory:** It resides on the GDRAM and its data bandwidth and rate is same as the global memory. This memory is put into use when too many registers are needed for an application where a large data structure or too many kernel variables exist.
- **Shared Memory:** It resides on the GPU chip and it has higher data bandwidth with lower latency characteristics compared to the local memory. Threads within the same block can communicate with each other via this memory. While using this memory, it is important to be careful about the bank conflict case in which multiple simultaneous accesses to a bank take place and hence, the overall process is serialized.
- **Global Memory:** It resides on the GDRAM and its access to the data can be

realized with different memory segments (32-, 64-, or 128-bit). To be able to get a high memory performance, data transfers can be realized preferably in an aligned manner and coalesced memory access is thus achieved.

- **Constant Memory:** It resides on the GDRAM and it can be utilized as a cache memory via the existing constant cache units. It is efficiently used when its usage includes single address. If several separate requests are taken, the throughput of this memory reduces proportionally.

3.2.2 CUDA Execution Model

The execution of a CUDA application is realized in hierarchical three levels as shown in Figure 3.6. These levels from the software perspective are thread, block and grid. At the lowest level, threads are executed by scalar processors on the GPU hardware. At the mid-level, thread groups constitute nonmigrant thread blocks and they are executed on the multiprocessors. Threads in a block can communicate with each other using the shared memory residing in their multiprocessor. Concurrency of several thread blocks is allowed on the same multiprocessor, but this might be limited by available resources like register space and shared memory. At the highest level, a collection of thread blocks form a grid, whose operation corresponds to a kernel launch in a CUDA program. Moreover, executing more than one kernel concurrently can also be realized in most of the current GPU devices.

Warp is another important concept in the execution model and it is defined as a parallelly executed thread group in the physical meaning. This works as a base unit in the thread scheduling operations of the multiprocessors. Each warp has 32 threads which are consecutively indexed. During the execution process, each block of threads with arbitrary size is further split into warp units. If the block size is not an integer multiple of 32, then some redundant (non-functional) threads will be padded to the end in order to satisfy 32 in the last part. The importance of the warp is that all threads in a warp *simultaneously* execute the same instruction for different data parts in accor-

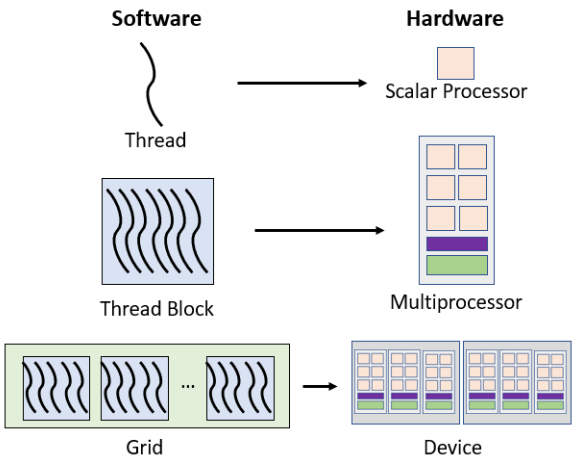


Figure 3.6: CUDA execution model.

dance with the SIMD (Single Instruction, Multiple Data) model. Providing enough work to the multiprocessor by requesting so many warps enables fast context switching and it is important to hide the latency occurring when an instruction waits for the result of a long operation.

3.2.3 CUDA Programming Model

In this subsection, some important basics for programming in CUDA will be mentioned in compliance with the memory and execution models previously given in Section 3.2.1 and Section 3.2.2, respectively. Basic programming steps in a typical CUDA application are illustrated in Figure 3.7.

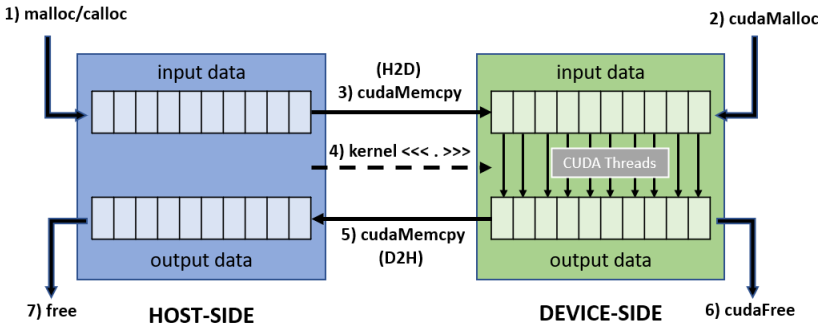


Figure 3.7: CUDA programming model.

Each step in this typical programming model is explained as follows.

1. The memory space required for the data to be transferred is allocated on the host memory via `malloc()` or `calloc()` function and the input data is prepared.
2. The device memory space to be used for the input data from host-side is allocated via `cudaMalloc()` function.
3. The input data is copied from host memory to device memory (through PCI-e bus) via `cudaMemcpy()` function in host-to-device (H2D) mode.
4. In order to launch all the threads in a grid to execute the defined work in a kernel function, kernel launch call is realized by the host side via `kernel<<<.>>>`
5. After all work is done by the CUDA threads and the resulting data is prepared, at the request of the host-side, the output data is copied from the device memory back to the host memory (again through PCI-e bus) via `cudaMemcpy()` function in device-to-host (D2H) mode.
6. When the transfer is finished, the memory space allocated on the device memory is released via `cudaFree()` function if the corresponding data is no longer to be used for any computation on the GPU.
7. If the related data is not necessary anymore, it is also a good practice to release the allocated memory space on the host-side via `free()` function.

The kernel function mentioned here can be seen as the heart of the computation done on the device-side (GPU). The function determines what each thread is responsible for doing and any interactions between them (if exists) are also defined inside it. As shortly mentioned in Section 3.2.2, the main execution structure of a kernel is based on a grid containing a certain number of thread blocks. A typical kernel call is realized by using some parameters that belong to the corresponding grid as follows.

```
kernel <<< blocks, threads >>> (...)
```

In the syntax given above, two commonly used parameters of a kernel function are stated, namely `threads` and `blocks`. According to the definition of the CUDA API [42], these two parameters can be specified in a type of either `int` or `dim3`. With the help of this, a 1D thread hierarchy can be directly constructed by using the parameters in `int` type and 2D/3D thread hierarchy can also be realizable by `dim3` type. The `threads` parameter is the total number of threads per block and it is commonly limited to 1024 threads[1]. The `blocks` parameter, on the other hand, is used to represent the total number of thread blocks within the grid.

Each CUDA thread which executes the kernel defined in the application is labeled with a unique thread ID that make it possible to access a specific thread within the kernel function, and this ID can be specified through a combination of kernel built-in variables. These variables are given with details in Table 3.3.

Table 3.3: CUDA built-in kernel variables [42].

Variable	Data Type	Functionality
`gridDim`	`dim3`	grid-structure dimensions
`blockIdx`	`uint3`	index of a block in the grid
`blockDim`	`dim3`	block-structure dimensions
`threadIdx`	`uint3`	index of a thread in the block
`warpSize`	`int`	number of threads in a warp

For the convenience of a CUDA programmer and better thread structure of a CUDA application, in addition to the 1D grid, it is also possible to construct 2D or 3D structures. In these cases, thread indexing has more than one dimension and the desired access is governed by combination of these indices. An example for 3D grid structure is given as follows.

[1] CUDA devices with compute capability > 2.0 support this value.

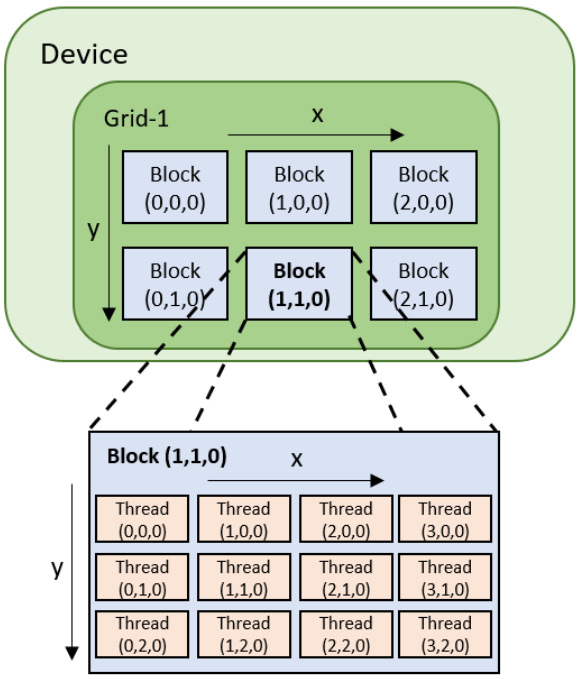


Figure 3.8: CUDA 3D-grid indexing example.

In Figure 3.8, a 3D grid structure of CUDA threads in an application is given, which is actually 2D since 3rd dimension (`z`) is never used here. First dimension of the structure is shown with `x` and the second one with `y`. A specific thread in this structure can be uniquely defined with indices `i` and `j`, preferably adding the index `k` which is always zero. These indices can be determined the expressions given in Table 3.4.

Table 3.4: CUDA thread indexing for the grids in different dimensions.

Grid	Thread Indexing
1D	`i = blockIdx.x * blockDim.x + threadIdx.x`
2D	`i = blockIdx.x * blockDim.x + threadIdx.x` `j = blockIdx.y * blockDim.y + threadIdx.y`

3D	`i = blockIdx.x * blockDim.x + threadIdx.x`
	`j = blockIdx.y * blockDim.y + threadIdx.y`
	`k = blockIdx.z * blockDim.z + threadIdx.z`

In CUDA programming, due to some software-based requirements, variables used in both the host- and device-side of the program may need to be used with different type qualifiers depending on the situation. These qualifiers with their properties are tabulated in Table 3.5.

Table 3.5: The variable type qualifiers in CUDA with their properties.

Variable Type Qualifiers	Memory	Scope	Lifetime
`__device__`	Global	Grid	Application
`__device__ __shared__`	Shared	Block	Block
`__device__ __constant__`	Constant	Grid	Application
`__volatile__`	-	-	-

Table 3.6: The function type qualifiers in CUDA with their properties.

Function Type Qualifiers	Executed in	Callable from
`__device__`	Device (GPU)	Device(GPU)
`__global__`	Device (GPU)	Host(CPU)
`__host__`	Host (CPU)	Host(CPU)

There are also some needs for qualifying the function types in host- and device-side of an application. These qualifiers are summarized in Table 3.6.

It is important to notice some points related to these qualifiers:

- Functions qualified by `__device__` and `__global__` cannot be used in recursive operations (not-supported).
- Qualifiers `__global__` and `__host__` cannot be used together.
- A function qualified with `__global__` needs to be defined with `void` return type.

3.2.4 CUDA-C/C++ Application

As a final point in the section related to CUDA, some brief information on the executable generation procedures for a C/C++ code (.cpp) and a CUDA code (.cu) is given. In Figure 3.9, a C/C++ project including CUDA codes is demonstrated from the source code phase to the executable/dll phase.

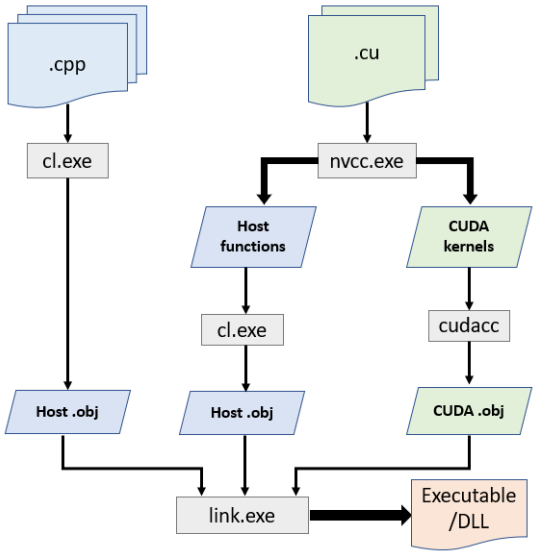


Figure 3.9: Executable generation process including .cpp and .cu codes.

As seen in the figure above, C/C++ codes (.cpp) only include the parts to be executed on the host (CPU). These codes can be compiled with gcc or cl.exe (MSVC) compiler. At the end of the compilation procedure, an object file is generated.

On the other hand, CUDA codes (.cu) include parts which will be executed on both host (CPU) and device (GPU). CUDA kernel functions with `global` identifier are called by CPU and executed on GPU. At the beginning of the compilation procedure, the CUDA compiler (nvcc) splits the overall code into host functions and CUDA kernel functions which will be executed on CPU and GPU, respectively [48]. Host functions in the CUDA code are assigned to a standard cpp compiler (gcc, cl.exe, etc.) and the CUDA kernel functions are assigned to cudacc compiler which is aware of all compilation rules (cuda.rules). When two sides are successfully completed, object files are generated for both host and device parts.

At the last step, all object files (.obj) are combined with a linker (link.exe) and a final executable(.exe) or a dynamic link library (dll) of the application is generated.

CHAPTER 4

IMPLEMENTATIONS

As mentioned in Section 2.4, this thesis is based on four noise subspace-based methods (PHD, MUSIC, EV, and MN) and these methods were implemented in three different programming environments: MATLAB, C/C++ and CUDA. The purpose for these implementations is to evaluate the methods in terms of numerical accuracy and computational performance. In this chapter, firstly, a pseudocode comparison of the four methods is given, and then the implementations in MATLAB, C/C++ and CUDA are mentioned in three separate sections with all relevant details.

4.1 Pseudocode Comparison

To understand the details behind the implementations, it is better first to compare the pseudocodes that belong to four noise subspace-based methods and determine their similarities and differences between them. The common code structure which is valid for all methods is given in Algorithm 1.

As can be seen in Algorithm 1, it takes some inputs, which are explained as:

- X represents the input data obtained from the antenna array outputs.
- d is the number of sources in the scenario, which is assumed to be known (see Assumption-4 in Section 2.3.2).
- $arrayConfig$ is the array-related information including number of antennas (m), their coordinates etc.
- f is the RF carrier frequency of the signal(s) transmitted by the emitter(s) located in the far field region of the receiving array (see Assumption-1 and Assumption-2 in Section 2.3.2).
- $azims$ represents the azimuth angles used in the spectral search and they are predetermined by start/stop/step angle values.
- $elevs$ represents the elevation angles, which are determined and used similarly in $azims$.

Table 4.1: Some variables used in Algorithm 1 and their definitions.

Variables	Definitions
X	m-by-N signal data from antenna array
d, f	number of emitters (known), their frequency
$arrayConfig$	array configuration parameters: m (# of antennas), coords
$azims, elevs$	azimuth/elevation angle series
$\widehat{R}$	m-by-m sample covariance matrix
$u, sing, v$	m-by-m SVD decomposition matrices
a_{ind}	m-by-1 array steering vector for indth angle pair
C	m-by-m projection matrix used in pseudospectrum

Algorithm 1 Common pseudocode for noise subspace-based DOA algorithms

1: **procedure** NSSDOA($X, d, arrayConfig, f, azims, elevs$)
2: $\widehat{R} \leftarrow X * X^H$ ▷ Step-1
3: $[u, sing, v] \leftarrow svd(\widehat{R})$ ▷ Step-2
4: $nssEig \leftarrow nssDetermination(.)$ ▷ Step-3
5: $C \leftarrow projProd(nssEig)$ ▷ Step-4
6: $angleSeries \leftarrow SeriesGen(azims, elevs)$ ▷ Line 6-12 : Step-5
7: **for all** $anglePair_{ind} \in angleSeries$ **do**
8: $a_{ind} \leftarrow steeringVector(anglePair_{ind}, f, arrayConfig)$
9: $a_{ind}^H \leftarrow adjoint(a_{ind})$
10: $invP_{ind} \leftarrow a_{ind}^H * C * a_{ind}$
11: $pseudoSpec(ind) \leftarrow P_{ind} \leftarrow (invP_{ind})^{-1}$
12: **end for**
13: $[peakVals, indices] \leftarrow findPeaks(PseudoSpec)$ ▷ Line 13-14 : Step-6
14: $estimDOA \leftarrow PeakSelection(peakVals, indices, d)$
15: **return** $estimDOA$
16: **end procedure**

After taking the aforementioned inputs, the common code structure in Algorithm 1 follows these steps:

- **Step-1 (line 2):** This step generates m-by-m $\widehat{R}$ (sample covariance matrix) from (m-by-N) X (array output matrix) and its Hermitian (N-by-m) where m is the number of antennas and N is the number of signal samples (snapshots).
- **Step-2 (line 3):** In this step, singular-value-decomposition (SVD) of $\widehat{R}$ is taken. Theoretically, this step should have been realized by eigen value decomposition (EVD). However, since $\widehat{R}$ is Hermitian and positive definite matrix [49], EVD

and SVD factorizations coincide [50]. Therefore, the SVD operation can be utilized without any problem. This operation here gives us three (m-by-m) output matrices ($u, sing, v$), which are left singular vectors, diagonal singular value matrix (in decreasing order) and right singular vectors, respectively.

- **Step-3 (line 4):** This step is for arranging how noise and signal subspaces are spanned by the corresponding eigenvectors. The determination of which eigenvector(s) are taken to lie in the noise subspace plays a critical role in the performance of our algorithms as explained later.

- **Step-4 (line 5):** This step applies an orthogonal projection operation onto the range space of the corresponding matrix/vector via a complex inner product and an (m-by-m) matrix C, which is a noise subspace representative is obtained at the end.

- **Step-5 (line 6 - 12):** This step can be referred to as "Spectral-search". In this step, an angle series is firstly created (Step-5.0) (see line 6) by considering all angle pair combinations of azimuth angles ($azims$) and elevation angles ($elevs$). Then the spectral search in $angleSeries$ is realized by following these sub-steps:

  - At Line 8 (Step-5.1), the steering vector (a_{ind}) is generated for the corresponding angle pair in the search using the signal frequency (f) and array coordinates (in $arrayConfig$).
  - At Line 9 (Step-5.2), the hermitian of the steering vector a_{ind}^H is obtained by taking an adjoint operation (i.e., transpose and then complex conjugate) of the steering vector (a_{ind}).
  - At Line 10 (Step-5.3), a product of three terms a_{ind}^H, C and a_{ind} is realized to measure the orthogonality of the corresponding steering vector and the noise subspace term C. When the resulting $invP_{ind}$ is close to zero, it could be implied that the orthogonality is somehow achieved.
  - At Line 11 (Step-5.4), the term $invP_{ind}$ is inverted, hence its small value in a case is converted into a large one and it is stored in $pseudoSpec$.

  All these sub-steps are repeated for each angle pair in $angleSeries$ and the vector $pseudoSpec$ is filled. This provides us with a 1D or 2D pseudospectrum showing some peaks.

- **Step-6 (line 13 - 14):** These two lines are basically for determining the peaks of the pseudospectrum obtained at the end of the spectral search and then selecting the certain number of these peaks according to the number of sources (d) which is assumed to be known at the beginning of the DOA estimation procedure. Finally, the estimation angle pair(s) corresponding to the selected peak(s) can be easily determined by tracing the relevant index information in $angleSeries$.

Table 4.2: Variables used in Algorithm 2 and their definitions.

Variables	Definitions
d	number of emitters (known)
u	m-by-m matrix containing left singular vectors
$sing$	m-by-m matrix containing singular values

Algorithm 2 Differences in nssDetermination (Step-3)

1: **procedure** PHDNSSDET($u, sing$)
2: $minIdx \leftarrow min(diag(sing))$
3: $e_{min} \leftarrow u.col(minIdx)$
4: **end procedure**

5: **procedure** MUSICNSSDET(u, d)
6: $E_n \leftarrow u.col(d+1 : end)$
7: **end procedure**

8: **procedure** EVNSSDET($u, sing, d$)
9: $E_n \leftarrow u.col(d+1 : end)$
10: $w_{dn} \leftarrow sing.diag(d+1 : end)$
11: $E_{n,w} \leftarrow E_n$ ▷ Initialize this with unity weights
12: **for** $i \leftarrow 1, cols(E_n)$ **do**
13: $E_{n,w}.col(i) \leftarrow E_n.col(i) * (w_{dn}(i))^{-1}$
14: **end for**
15: **end procedure**

16: **procedure** MNNSSDET(u, d)
17: $E_n \leftarrow u.col(d+1 : end)$
18: $P_n \leftarrow E_n * adjoint(E_n)$
19: $unitV \leftarrow [1, 0, ..., 0]_M$
20: $\lambda \leftarrow (adjoint(unitV) * P_n * unitV)^{-1}$
21: $valph \leftarrow (P_n * \lambda * unitV)$
22: **end procedure**

Even if four methods in our scope (PHD, MUSIC, EV, and MN) share a common code structure as in Algorithm 1, it is important to notice that there are also some critical differences between them. These differences basically happen in Step-3 (line 4) and Step-4 (line 5) of Algorithm 1. These differences are given in Algorithm 2 and Algorithm 3 with 1-based indexing format.

As can be seen in Algorithm 2, PHD determines the noise subspace only with a single eigenvector that corresponds to the smallest eigenvalue and this eigenvector is referred here as e_{min}. On the other hand, MUSIC determines the noise subspace by using all $(m - d)$ eigenvectors corresponding to the smallest $(m - d)$ eigenvalues. These eigenvectors are put into the columns of the matrix E_n. EV determines the

noise subspace similarly to MUSIC. Instead of keeping directly the noise eigenvectors for Step-4 operation, it keeps a modified version of E_n, which is obtained by weighting each noise eigenvector with the inverse of its corresponding eigenvalue. As opposed to MUSIC and EV methods, MN method determines the noise subspace using a single vector with special properties. This single vector in MN has a minimum norm and its first element should satisfy unity.

Table 4.3: Variables used in Algorithm 3 and their definitions.

Variables	Definitions
e_{min}/v_{alph}	selected noise subspace (nss) single-vector (PHD/MN)
$E_n/E_{n,w}$	selected noise subspace (nss) matrix (MUSIC/EV)
C	projection matrix obtained by using nss vector/matrix

Algorithm 3 Differences in projProd (Step-4)

procedure PHDPROJPROD(e_{min})
    $C \leftarrow e_{min} * adjoint(e_{min})$
end procedure

procedure MUSICPROJPROD(E_n)
    $C \leftarrow E_n * adjoint(E_n)$
end procedure

procedure EVPROJPROD($E_n, E_{n,w}$)
    $C \leftarrow E_{n,w} * adjoint(E_n)$
end procedure

procedure MNPROJPROD($valph$)
    $C \leftarrow valph * adjoint(valph)$
end procedure

As can also be seen in Algorithm 3, PHD and MN use the orthogonal projection onto the range space of a single vector (e_{min} or $valph$). On the other hand, MUSIC and EV apply the orthogonal projection for matrices which carry more information than a single vector.

All differences mentioned above basically determine the estimation accuracy performance of four algorithms. Strong and weak sides of each algorithm take roots from these differences.

4.2 MATLAB Implementations

In order to initially evaluate four algorithms in our scope (PHD, MUSIC, EV, and MN) from the estimation accuracy point of view, all were implemented in MATLAB following the structure mentioned in Section 4.1. All data structures used in the codes are in double precision (FP64) by default. Within all these implementations, only standard built-in functions of MATLAB were used and no special toolbox function is utilized in the codes. In this thesis, these implementations are used for following purposes:

- To evaluate the estimation accuracy performance of algorithms under different scenario parameters (array structure, aperture, SNR-level, and number of direct paths) (see Section 5.1)
- To create a ground truth for numerical accuracy evaluation of the implementations in C/C++ and CUDA (see Section 5.2.2)
- To create a baseline for computational (time) performance of C/C++ and CUDA implementations (see Section 5.2.4)

As would naturally be expected, in order to prepare a working code flow and evaluate the results in a controlled way, we need real or synthetic input data with known signal/scenario parameters. Since it is difficult to arrange controlled experiments and collect real scenario data, a simulated data solution was preferred in the analyses. For this, a proprietary synthetic signal generator developed at ESEN System Integration Ltd. was used to prepare input data in MATLAB for different test scenarios. To perform the experiments in appropriate manner, all implementations (MATLAB, C/C++ and CUDA) have utilized the same input data that was synthetically generated.

The synthetic signal generator used in our experiments is a MATLAB code that simulates an RF transmission/reception scenario and data collection by a multichannel phase-coherent RF receiver system. This code can simulate a certain number of emitters as if their signals impinged on an antenna array (in a certain structure and aperture size) with direct path(s) and possibly multipath propagation(s). This generator enables to create signals with the following parameters:

- Sampling Frequency (Fs)
- Number of Samples (N)
- Incoming/impinging signal (carrier) frequency (f)
- Direct path (DP) number
- Multipath (MP) number
- Direct path/Multipath DOA angles (in azimuth and elevation)
- Signal amplitude and type (sinusoidal, random etc.)
- Noise type (AWGN or randn)

- Signal-to-noise ratio (SNR)
- Number of antennas (m)
- Antenna array structure/geometry (linear, circular, random etc.)
- Antenna array configuration (coordinates or for special geometries: radius, element spacing, etc.)
- Multipath-related parameters (delay, correlation bandwidth etc.)

4.3 C/C++ Implementations

In the C/C++ implementations of four noise subspace DOA methods (PHD, MUSIC, EV and MN), mostly *Eigen library* [51] is utilized in the codes to construct the MATLAB-equivalent data structures and also to use the equivalent functions. Eigen library is a C++ template library, which means that the project has a dependency on Eigen in compile-time only. Because of this, it is simply required to include the library directory in the project settings and there is no need to follow a long installation procedure. Therefore, using this library in a project requires very little effort. The Eigen library supports many dense and sparse linear algebra utilities, some of which are given as:

- Different data-structures (matrix, array, row-vector, etc.)
- Mapping of the external arrays
- Various dense/sparse matrix manipulations
- Several reduction methods (min/max, sum, norm, etc.)
- Sub-matrix accesses (row, column, segment or block operations)
- Different decomposition methods (LU, QR, JacobiSVD, etc.)
- Different geometrical operations (scaling, translation, 2D/3D rotation, transform, etc.)

Eigen is also known for providing vectorization activated by telling the compiler (gcc, MSVC, clang, etc.) to enable the relevant instruction set. The vectorization is simply the preparation of a program to operate on a vector processor via SIMD instructions. In general, the vectorization process can be achieved manually or automatically by the compiler. In our case, it is done automatically. The Eigen library supports most of the well-known SIMD instruction sets (AVX, SSE, AltiVec, etc.) of different architectures [51]. The data-level parallelism coming from the SIMD instructions in Eigen is realized by doing the operations simultaneously on the vector elements and this provides an important performance improvement in the codes.

As mentioned in Algorithm 1, other than the standard matrix/vector operations in the code flow, we have a singular-value-decomposition (SVD) in Line 3. It is a special

operation since the decomposition applied on the spatial covariance matrix $\widehat{R}$ gives us a list of eigenvectors, some of which determine the noise subspace and hence the estimation performance of our DOA methods. The SVD operation in the C/C++ implementations is realized by using *JacobiSVD* method [51] of the Eigen library. This method is based on two-sided Jacobi singular value decomposition of any rectangular matrix and it is known to have optimal accuracy and reliability. By default, only singular values are computed by the method and in order to get left and right singular matrices, they need to be explicitly requested within the argument.

In order to determine the most time-consuming parts in the algorithms, since they are quite similar, the MUSIC serial C/C++ code was chosen for the hotspot analysis. In this analysis, a spectral search was realized in azimuth=[0:1:359](deg) and elevation=[0:1:90](deg) ranges, and the execution of the serial MUSIC algorithm was repeated enough times (~1000) for reliability. At the end, the average duration for each part was determined and the corresponding time distribution is profiled as follows.

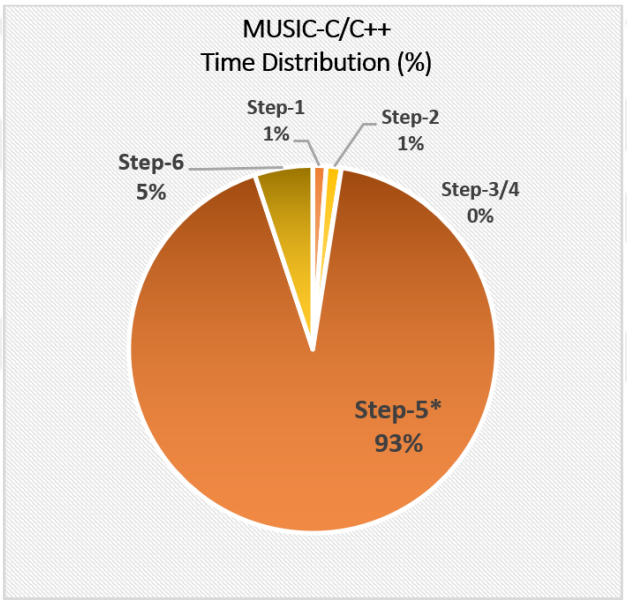


Figure 4.1: Time distribution for serial MUSIC algorithm in C/C++.

As can be seen in Figure 4.1, there are two primary hotspot parts, which are namely Step-5* and Step-6. Step-5* is apparently recognized as the most time-consuming part dominating the overall distribution (93% of the whole execution time). It is simply defined as Step-5 as a whole in Algorithm 1 excluding the computation of the array steering vector (see Line 8). In this step, a spectral search is realized to completely fill a pseudospectrum until the end of the loop. On the other hand, Step-6 is the second most time-consuming part (5% of the whole execution time) and as can be seen in Algorithm 1, it includes finding peaks in the pseudospectrum and selecting certain ones for DOA estimation (see Line 13 and Line 14).

The results obtained by the hotspot analysis given above can also be verified analytically by the computational complexity analysis for the MUSIC algorithm. This analysis is tabulated in Table 4.4.

Table 4.4: Total number of computations at each step of the MUSIC algorithm.

Step Number	Total Computation
Step-1	$(3m^2N + mN)/2$
Step-2,3	$12m^3$
Step-4	$m^3 + (1/2 - d)m^2 - (1/2 + d)m$
Step-5*	$L(2m^2 + m)$
Step-6	$L\ log(L)$

In the Table 4.4, the complexity parameters m, N, d, L represent the number of antennas, the signal sample (snapshot) number, the number of incoming signals and the number of angle pairs in the spectral search (scan), respectively. In our scenario, all variables except L take relatively small values. However, since L represents the angle scan range, it generally takes a large value to cover all possible angle pairs in the DOA estimation. As can be seen, Step-5* and Step-6 have a multiplicative term L and this actually explains why these two steps dominate the overall execution time in Figure 4.1.

After determining the hotspots and verifying with complexity analysis, these parts are further evaluated with respect to their parallelization potential. Following Algorithm 1, in the C/C++ implementations, Step-5* was realized within a "for-loop" over angle scan range (L) and there is no data dependency between indices in this loop. Therefore, a data-level parallelism is applicable for this part. On the other hand, Step-6 includes a function related to finding all peaks in the resultant pseudospectrum and this part can also be parallelized at a certain level.

In order to construct a parallelization basis for the upcoming CUDA implementations and create a baseline for fair performance measurements, Step-5* and a part of Step-6 were parallelized by using OpenMP. This enabled us to realize multi-threaded code execution for these parts on CPU. Some brief implementation details are explained as follows.

- **The parallelization of Step-5* (spectral-search) via OpenMP:**

  As can be seen in Figure 4.2, two innermost "for" loops starting at line-7 and line-9 with indices i and j are enclosed with two OpenMP directives. The outermost one informs the compiler about that an OpenMP parallel region needs to be initialized with certain number of threads ($ThreadNum$) and some variables ($a, i, j, etc.$) should be created for each thread using thread private memory (see Section 3.1.4). The second OpenMP directive, on the other hand, is used to inform the compiler about how the parallelization is realized. This directive includes the request about parallel for-loop where threads are dynamically scheduled. This setting is preferred since there is no particular order in which the chunks are distributed to the threads and by this, a higher efficiency can be achieved [39].

```
4   #pragma omp parallel private(a,a_adj,i,j,invP) num_threads(ThreadNum)
5   {
6       #pragma omp for schedule(dynamic)
7       for (i = 0; i < n_azim; i++)
8       {
9           for (j = 0; j < n_elev; j++)
10          {
11              a = arrayManifold[i].col(j);
12              a_adj = a.adjoint();
13              invP = a_adj * C * a;
14              P(i, j) = 1 / (invP.real());
15          }
16      }
17  }
```

Figure 4.2: MUSIC C/C++ code snippet for Step-5* parallelized via OpenMP pragmas.

- **The parallelization of findPeaks() in the Step-6 via OpenMP:**

```
20  #pragma omp parallel private(i,j)  num_threads(ThreadNum)
21  {
22      #pragma omp for
23      for (i = 0; i < max(colSize, rowSize) - 1; i++)
24      {
25          ...
26      }
27          ...
28      #pragma omp for
29      for (i = (1 - isOneRow); i < rowSize; i++)
30      {
31          for (j = (1 - isOneCol); j < colSize; j++)
32          {
33              ...
34              {
35          #pragma omp critical
36                  {
37                      ...
38                  }
39              }
40          }
41      }
42  }
```

Figure 4.3: C/C++ block structure for findPeaks() parallelized via OpenMP pragmas.

As shown in Figure 4.3, the parallel region (line 20) is initialized similarly as done in Step-5* (Figure 4.2) with only two private variables this time. The execution of two for-loops (starting at line-22 and line-28) inside the parallel region is declared to be parallel via `#pragma omp for`. There is also a need for a block in the second for-loop to update a shared variable by a single thread at a time, which was achieved by `#pragma omp critical` and it prevents a possible race condition [41].

4.4 CUDA Implementations

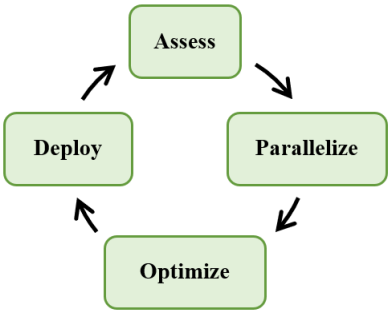


Figure 4.4: The CUDA design cycle (APOD).

In a typical CUDA implementation, a design cycle proposed by NVIDIA© is generally applied. This cycle is visually represented in Figure 4.4 and it mainly consists of the following four steps [47]:

- **Assess:** In most of the applications, this is the first step and it includes a process in which the parts of the code taking up the majority of the execution time are located. The process can also be referred to as bottleneck or hotspot analysis. By this analysis step, the hotspots in the code can be further evaluated for parallelization and potential GPU acceleration.
- **Parallelize:** After the assessment step, problematic (in terms of time performance) but parallelizable parts are determined and these parts are focused by the developer. Depending on the complexity and the applicability of the original code, some existing libraries which are already GPU-optimized (e.g., Thrust, cuSOLVER, and cuBLAS) can simply be used, or it is sometimes necessary to restructure (refactor) the code at a certain level to benefit from the inherent parallelism. In general, the main aim here is to realize the parallelism in a simple way while harnessing the power of the GPUs as efficient as possible.
- **Optimize:** After each possible parallelization attempt, an optimization procedure can be applied by the developer. For some scenarios, there could be several possible optimizations that are worth considering. In this situation, the optimization procedure can be realized in an iterative way. There is also a trade-off between the effort spent for an optimization and the performance improvement that it can provide. Because of that, applying more reasonable optimizations is generally preferred.
- **Deploy:** This step is generally applicable in a code development scenario involving an end-product at the final stage. Deploying an application on the target system at a relatively early stage is useful for observing a potential speedup at even a preliminary level and it helps to reduce some possible risks in the code development process.

After completing the hotspot, the computational complexity, and preliminary performance analyses for C/C++ implementations as mentioned in Section 4.3, the most time-consuming parts and their suitability for parallelization were determined. This procedure actually corresponds to "Assess" in the CUDA design cycle (Figure 4.4) and this was generally adopted for the CUDA implementations of the algorithms in this thesis.

For the parallelization of the serial C/C++ codes (second step in the design cycle), the analysis results obtained from Section 4.3 were used. Since most of the execution time (93 %) is devoted to Step-5*, at the first stage, this step is evaluated from the parallelization point of view.

Apart from Step-5*, some additional parts are also investigated to determine their suitability for implementing on the GPU. One of the attempts in that scope was to look for the possibility of an efficient SVD computation on the GPU even if its contribution to the overall performance is expected to be small. Nevertheless, the SVD implementation on the GPU was realized by utilizing cuSOLVER-gesvdj method and the implementation details were adapted from an example code provided officially by NVIDIA© [52]. The gesvdj method is provided under cuSOLVER, which is known as a GPU-accelerated library whose aim is to provide the CUDA users with some LAPACK-like features (e.g., matrix factorization and eigenvalue solver).

All CUDA codes were generally adapted from the C/C++ codes and therefore, the computations on the host (CPU) are mostly realized by using Eigen library operations and functions. Moreover, *cuComplex wrapper* was used for all complex-valued computations on the GPU.

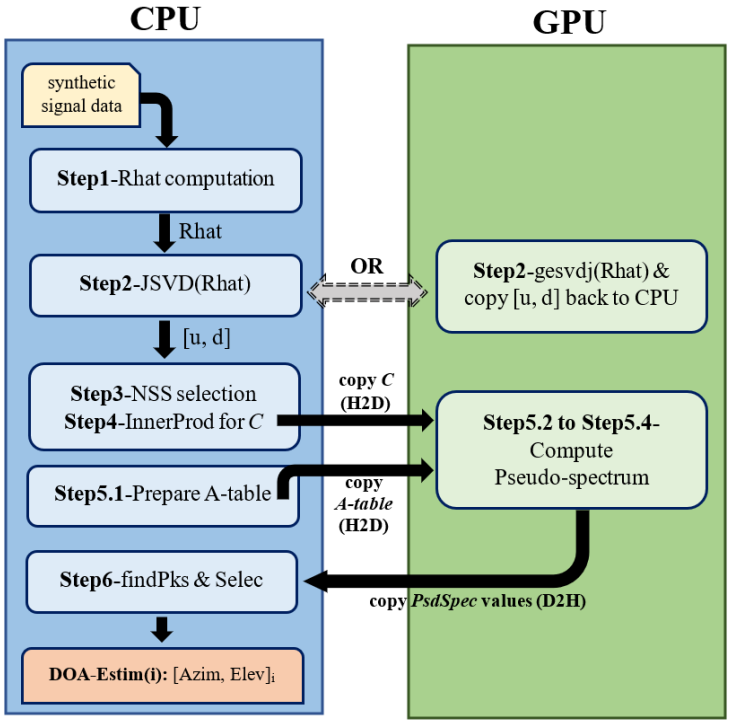


Figure 4.5: Heterogeneous (CPU-GPU) structure for the CUDA implementations.

Taking all implementation details into account, the structure in Figure 4.5 was created considering all algorithms in our scope to realize a CPU-GPU heterogeneous computation. This structure can be explained step-by-step as follows:

- **Step-1:** R_{hat} computation is done on the "**CPU**" using the complex-valued signal matrix X (and copying to GPU memory if necessary)
- **Step-2:** The SVD operation for R_{hat} is realized by one of the following:
  - The cuSOLVER-gesvdj method can take SVD on the "**GPU**" and the resulting complex-valued matrices should be copied back to the CPU for the further steps.
  - The JacobiSVD method of Eigen library can take SVD on the "**CPU**" and its complex-valued results can be used directly for the further steps.
- **Step-3:** The noise subspace span is determined and some operations (e.g., weighting, normalization, etc.) are done on the complex eigenvectors according to the approach adopted by the corresponding DOA method (PHD, MUSIC, EV or MN). This step is done on the "**CPU**".
- **Step-4:** The calculation of complex inner product is realized on the "**CPU**", which is done differently for each method and a complex-valued matrix C is obtained at the end. The resulting matrix is put into a host array (C_h) and its content is copied from RAM (host) to global (device) memory.
- **Step-5:** This step consists of the following sub-steps:
  - **Step-5.0:** A vector containing all possible angle pairs is generated on "**CPU**" using $azims$ and $elevs$ scan ranges.
  - **Step-5.1:** The steering vectors for all generated angle pairs are computed on "**CPU**" and they are kept in a lookup-table array (A_table_h). This array is copied from host memory (RAM) to global (device) memory and each steering vector partition in the array (A_table_d) is used by a separate CUDA thread on the GPU.
  - **Step-5.2, Step-5.3 and Step-5.4:** These include the computation sub-steps starting with input arrays C_d and (A_table_d), resulting with the (P_table_d) array including the pseudospectrum values. All sub-steps are defined in a CUDA kernel function and they are executed for each angle pair on a separate CUDA thread which works in parallel with the other ones. The parallelization is here based upon the different array steering data partitions of the (A_table_d) array and the SPMD (single program, multiple data) approach is hence adopted in the parallelization procedure.

    As given in Figure 4.6, a typical kernel used in our CUDA implementations is configured with `BLKSIZE` and `blNum` parameters. The number of threads spawned in a block is represented by `BLKSIZE` and `blNum` is used to represent how many blocks are required in a task. Total number of active threads is determined by the parameter named as `TotalScan` and it depends on the angle scan ranges ($azims$ and $elevs$). Due to the

nature of thread organization in the CUDA environment, a certain number of threads are redundant, which means that they are spawned to complete a block, but not used in the computations. The number of these redundant threads in our case is computed by `(blNum*BLKSIZE-TotalScan)`.

```
/*  MUSIC kernel-configuration */
int BLKSIZE = 32;
int TotalScan = n_azim * n_elev; // total number of active threads to be spawned
int blNum = (TotalScan) / BLKSIZE + (TotalScan % BLKSIZE == 0 ? 0 : 1);


/* MUSIC kernel-launch with preferred configuration */
MUSIC_kernel <<< blNum, BLKSIZE >>> ( C_d, A_table_d,invPrhs_table_d,
                                      invP_table_d, P_table_d,
                                      n_azim, n_elev, TotalScan);

```

Figure 4.6: MUSIC-CUDA code snippet for the thread settings and kernel launch.

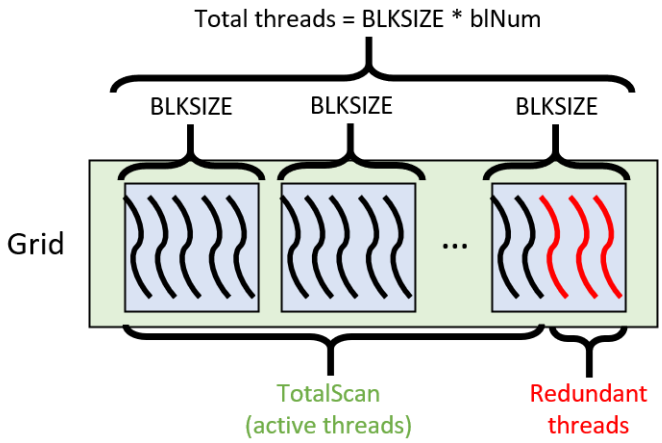


Figure 4.7: The typical thread organization used in CUDA implementations.

To explain better, the thread organization mentioned here is also visualized as in Figure 4.7. Step-5 is completed when all CUDA threads fill (P_table_d) array which contains the pseudospectrum values corresponding to scanned angle pairs. This completed array is then copied back to the CPU for being processed in the next step.

- **Step-6:** This is the last step in a typical implementation of the algorithms and this includes the following two sub-steps:
  - **Step-6.1:** After the pseudospectrum values are copied to (P_table_h) residing on the host memory, significant peak values are determined via 1D or 2D search, depending on how the initial angle scan is configured (1D: scan only in one of the angles or 2D: scan in both azimuth and elevation).
  - **Step-6.2:** After determining all the considerable peaks in the pseudospectrum, the largest d peaks are selected after a simple sorting operation and the corresponding angle pairs are determined. These angle pairs are nothing but d DOA estimation results of our algorithms.

CHAPTER 5

EXPERIMENTS AND RESULTS

There are two main types of experiments that were realized within the scope of this thesis, namely:

- DOA estimation accuracy-based theoretical experiments
- Implementation-based numerical and performance experiments

All the details related to the experiments and the corresponding results are mentioned in this chapter.

5.1 DOA Estimation Accuracy Analyses

The analyses to be mentioned in this section mainly include how the DOA estimation accuracy is affected depending on some system-based parameters, namely; array geometry (structure), array aperture and SNR-level of the impinging signal(s). All the experiments have been implemented using MATLAB and evaluated from the theoretical perspective. External effects that may be encountered in a real scenario were ignored in our analyses.

5.1.1 Effects of the array geometry

Array geometry (structure) is an important parameter in terms of the DOA estimation performance. It is simply determined by the arrangement of the array elements. As mentioned in Section 2.3.1, array geometries are classified into linear (1D), planar (2D) and volumetric (3D) arrays. Each category has also different subcategories determined by the spacing arrangement like uniform, non-uniform or random [53]. Some examples for array geometries are ULA (uniform-linear-array), UCA (uniform-circular-array), concentric circular and non-uniform L-,X-, or U-shaped structures. In our study, two common array geometries (ULA and UCA) were taken into consideration and their DOA estimation performances were comparatively investigated.

Firstly, two arrays ULA and UCA configurations were evaluated under d=2 (two sources, no multipath) case where the inter-element spacing is 10m for both arrays, the frequencies of the emitter signals are 15MHz, SNR level is at 20dB and two sources are coming at $[80°]$ and $[240°]$ in azimuth and at the constant elevation $[90°]$. For the pseudospectrum scan ranges, azimuth=[0:1:359](deg) and constant elevation=[90](deg) were chosen. DOA estimation plots of four algorithms (PHD, MUSIC, EV, and MN) for ULA and UCA structures are given in Figure 5.1 and Figure 5.2, respectively.

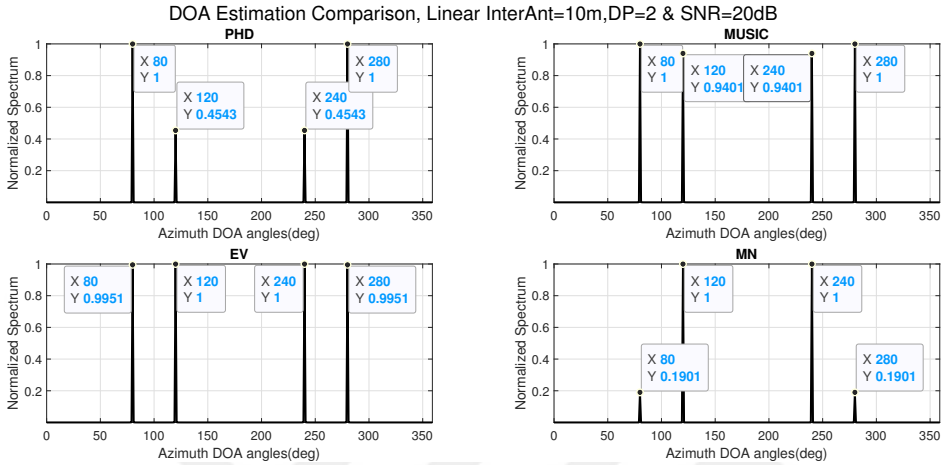


Figure 5.1: DOA estimation of four algorithms in the **ULA case** with d=2.

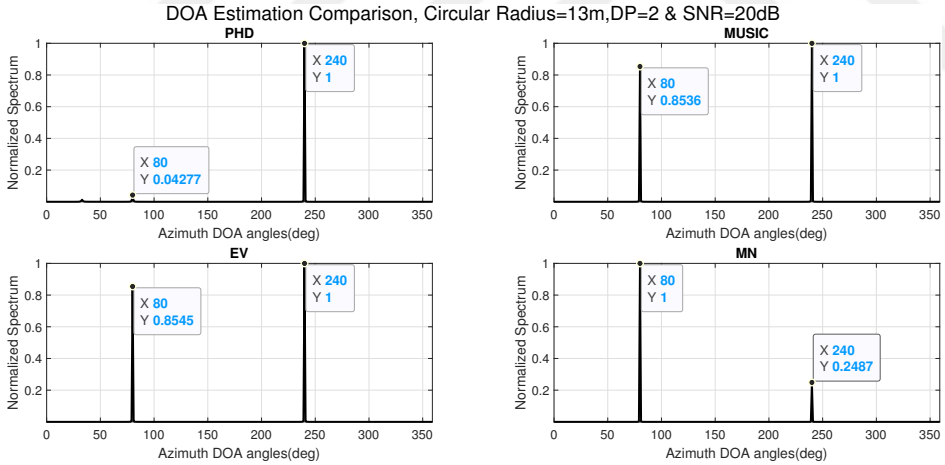


Figure 5.2: DOA estimation of four algorithms in the **UCA case** with d=2.

As can be seen from Figure 5.1, in the ULA case, each method has given two additional peaks other than the correct DOA angles ($[80°]$, $[240°]$). However, in the UCA case (see Figure 5.2), all four methods have successfully estimated the correct DOA angles and there are no additional peaks in their pseudospectrums.

In order to understand the reasons behind these additional peaks, the ULA case was studied under d=1 case (single source and no multipath) with the same scenario parameters. DOA estimation plots of four algorithms were obtained for two different scenarios with the ULA configurations, which are one with an emitter coming at $[80°]$ in azimuth and another one at $[240°]$. The corresponding results are given in Figure 5.3 and Figure 5.4, respectively.

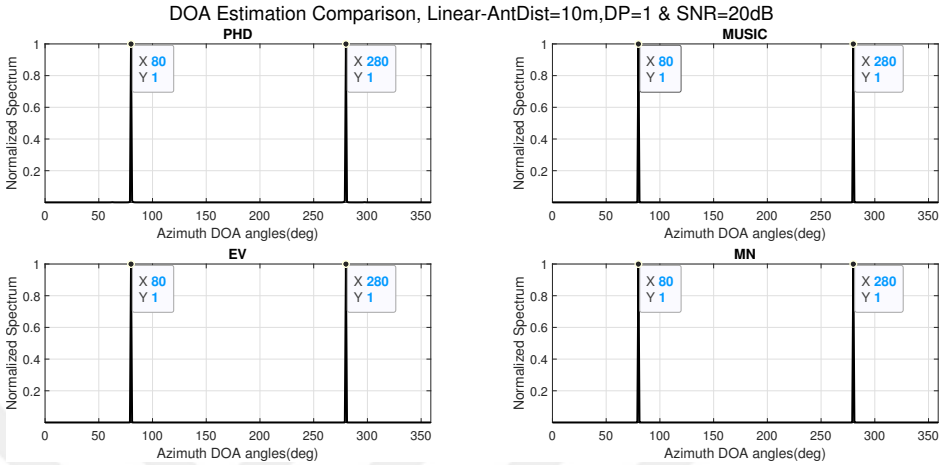


Figure 5.3: DOA estimation of four algorithms in the ULA case with d=1 at $[\mathbf{80°}]$ in azimuth.

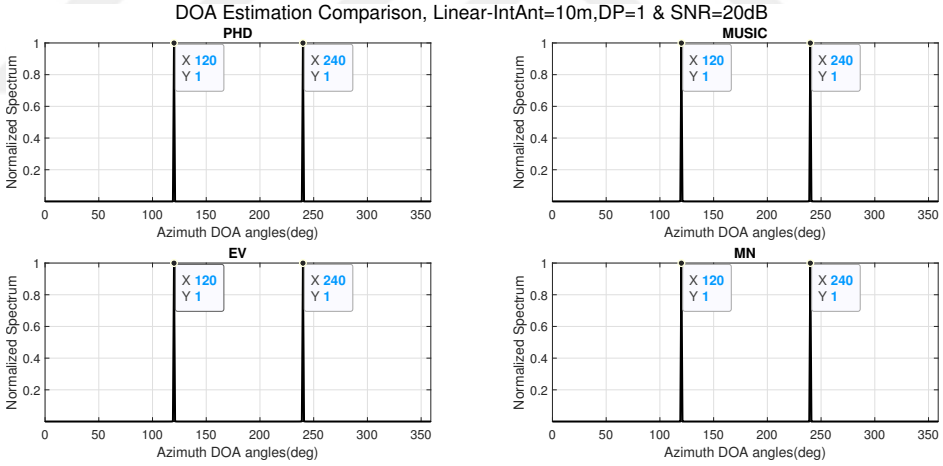


Figure 5.4: DOA estimation of four algorithms in the ULA case with d=1 at $[\mathbf{240°}]$ in azimuth.

As can be seen from Figure 5.3 and Figure 5.4, additional angle peaks at $[280°]$ and $[120°]$ have been obtained, respectively. These are nothing but symmetry values of the original DOA azimuths with respect to $[180°]$. This situation is known as "north-south ambiguity" which occurs when two wavefronts come from the north and the south, symmetrically [30]. Therefore, the DOA methods used with the ULA sensor system could not distinguish the half-space of the actual DOA of the impinging signals and they have given us additional peaks at the symmetrical side. Since the UCA covers all the planar space for all DOA angles, no such situation was observed.

5.1.2 Effects of the aperture size

Array aperture is another important parameter in DOA estimation performance and it can be actually derived from the array geometry together with the element spacing information. As mentioned previously in Section 2.3.1, the aperture of a (passive) sensor array can be defined as the spatial region occupied by the array. There is a direct relationship between the array aperture size and spatial resolution level. This relationship is that the aperture of an array determines the resolution with which two impinging wave fronts could be separated. Angular (spatial) resolution can be approximately calculated by $\theta_{angRes} \approx \lambda / ApertureSize$.

The effective apertures for the ULA and the UCA are significantly different from each other. The one for the ULA depends on the azimuth angle of the wavefront whereas there is no such situation for the UCA and it is basically equal to the diameter [30]. For the sake of a better observability, the UCA was mainly considered in our experiments based on different aperture sizes. In order to observe the corresponding effects on the DOA estimation, four circular array radii (1m, 3m, 5m and 10m) were investigated in d=2 case where the signals coming at $[150°]$ and $[200°]$ in azimuth with the frequency of 15 MHz at SNR level of 10dB. For the pseudospectrum scan ranges, azimuth=[0:1:359](deg) and constant elevation=[90](deg) were chosen again. The DOA estimation results corresponding to the different array radii are given in Figure 5.5 to Figure 5.8.

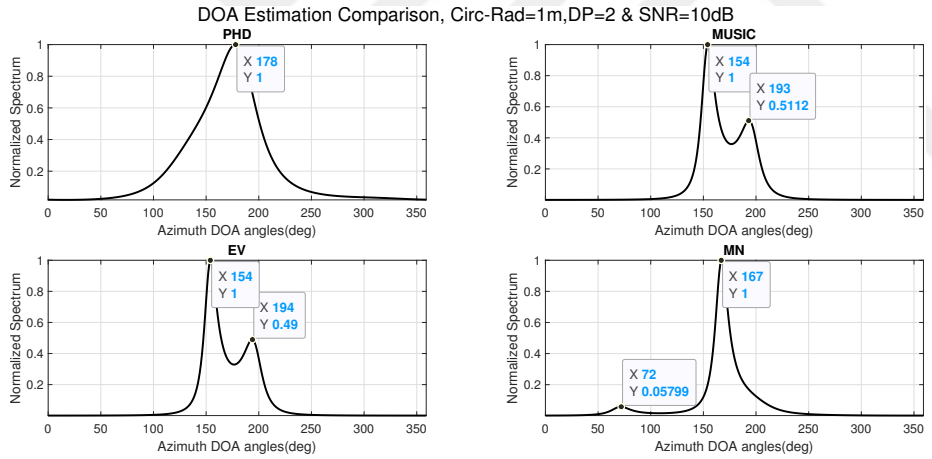


Figure 5.5: DOA estimation results in the UCA case with **rad=1m** and d=2 at $[150°]$ and $[200°]$.

As can be seen from the Figure 5.5, PHD and MN methods perform poorly in rad=1m case since the PHD spectrum has only one peak (there is even no sign for the number of emitter signals) and MN cannot resolve angles close to the expected DOA angles. MUSIC and EV methods have relatively better performance by making a maximum $7°$ DOA error. This performance difference between methods may occur since PHD and MN use only one vector to span their noise subspace and hence using less information. In the rad=3m case, all methods perform better compared to the rad=1m case. PHD can now resolve two angles with a maximum $11°$ DOA error and MN can resolve one of the angles with $5°$ DOA error. The DOA error for MUSIC and EV drops to $1°$. In the rad=5m and rad=10m cases, all methods have similar performance with a maximum $1°$ DOA error.

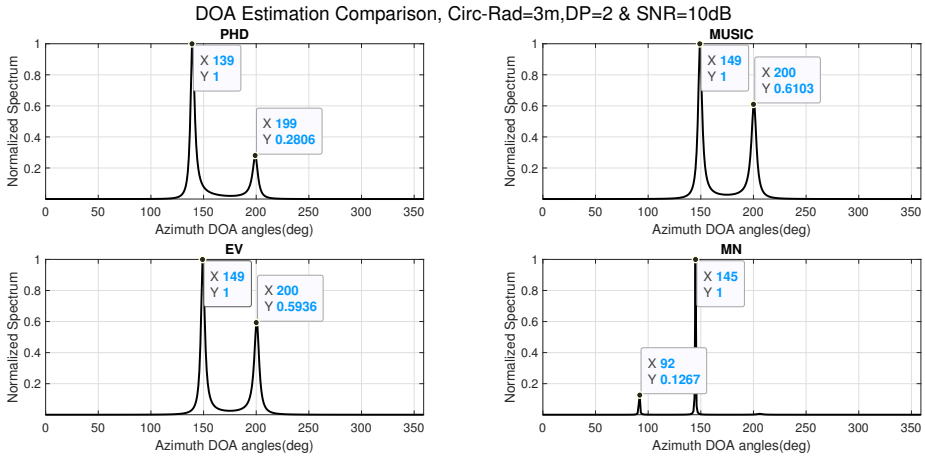


Figure 5.6: DOA estimation results in the UCA case with **rad=3m** and d=2 at $[150°]$ and $[200°]$.

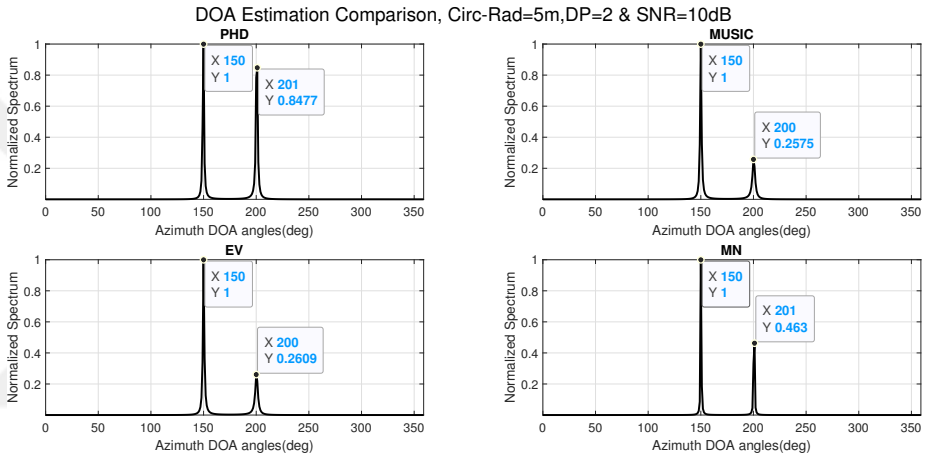


Figure 5.7: DOA estimation results in the UCA case with **rad=5m** and d=2 at $[150°]$ and $[200°]$.

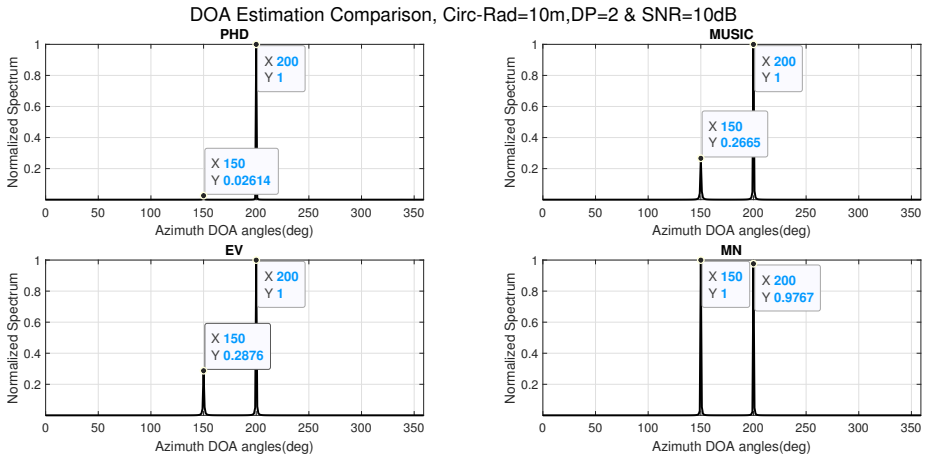


Figure 5.8: DOA estimation results in the UCA case with **rad=10m** and d=2 at $[150°]$ and $[200°]$.

5.1.3 Effects of SNR level

SNR (signal-to-noise ratio) is defined as the ratio of the signal power to the noise power and it is generally measured in decibel (dB) by taking $10\log_{10}$ of the power ratio. It is known that the DOA estimation performance is controlled by the accuracy of the estimated (sample) covariance matrix $\widehat{R}$ whose quality depends on the SNR [54]. Therefore, if we take signals from the array elements at a high SNR level, then it will be more probable to have a satisfying performance. In order to observe the effects of different SNR levels, the methods were studied under four SNR levels (-15dB, -5dB, 5dB and 15dB) in two different emitter cases (d=1 and d=2).

d=1 case

Firstly, the DOA estimations of the methods for d=1 case were investigated where single emitter is coming at $[160°]$ with the frequency at 15 MHz and UCA radius of 10m. For the pseudospectrum scan ranges, azimuth=[0:1:359](deg) and constant elevation=[90](deg) were chosen again. The DOA estimation results under four SNR levels are given in Figure 5.9 to Figure 5.12.

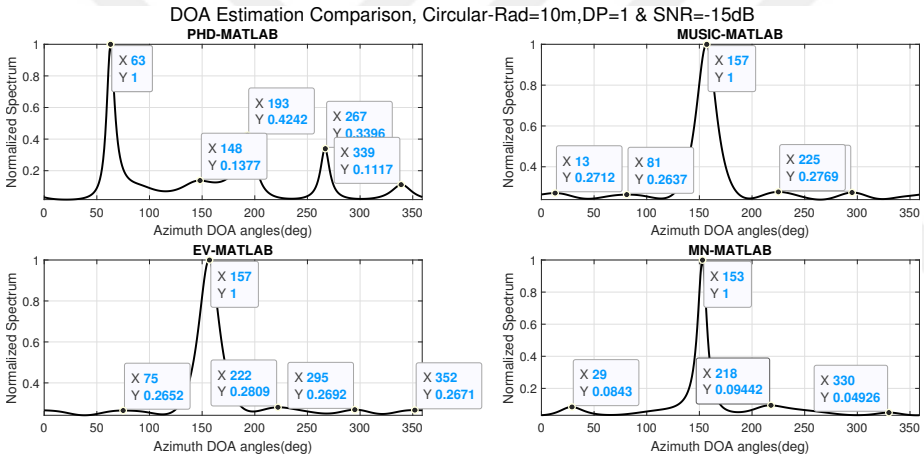


Figure 5.9: DOA estimation results in the UCA case under **SNR = -15dB** with d=1 at $[160°]$.

As can be seen from Figure 5.9, since SNR level is relatively low (-15 dB), all the DOA methods have difficulty in finding the single correct orthogonality condition for signal and noise subspaces since signal components are somehow lost within the noise component. Nevertheless, the performances of the methods (except PHD) are quite satisfying only with spurious peaks and $3° - 7°$ DOA error.

As can be seen in Figure 5.10, SNR = -5dB is an acceptable level for all the methods except PHD. Only the PHD method gives a result with $20°$ DOA error and all the others have zero error. As mentioned previously in Section 2.4.1, the DOA performance of the PHD method is susceptible to noise. The results presented here under low SNR values (-15dB and -5dB) confirm this claim. Under high SNR values (5dB and 15dB), as in Figure 5.11 and Figure 5.12, all methods including PHD perform perfectly such that no DOA error is observed in any one of them.

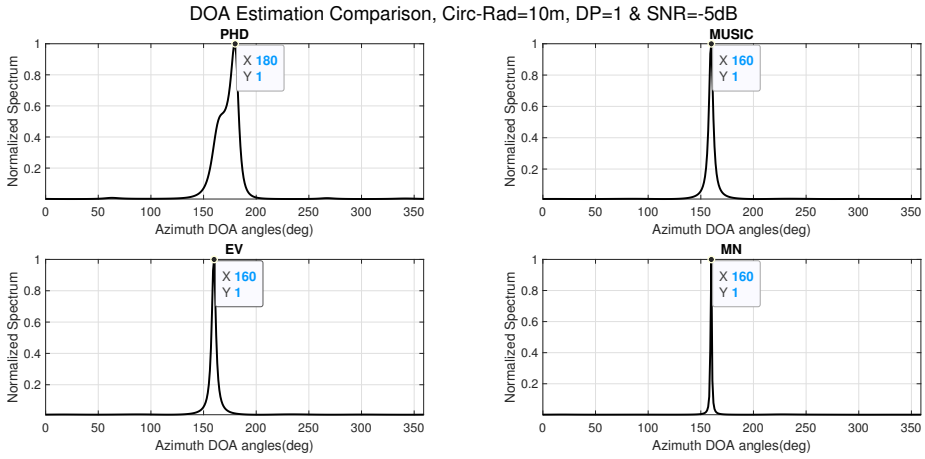


Figure 5.10: DOA estimation results in the UCA under **SNR = -5dB** with d=1 at $[160°]$.

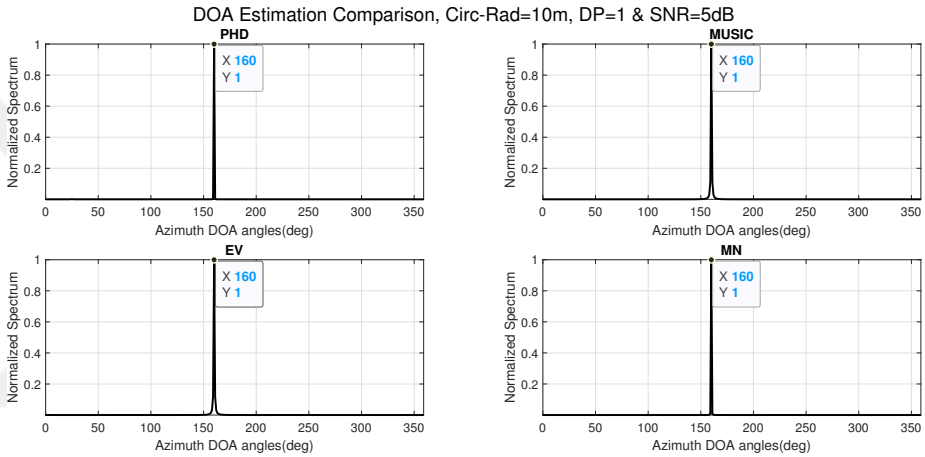


Figure 5.11: DOA estimation results in the UCA under **SNR = 5dB** with d=1 at $[160°]$.

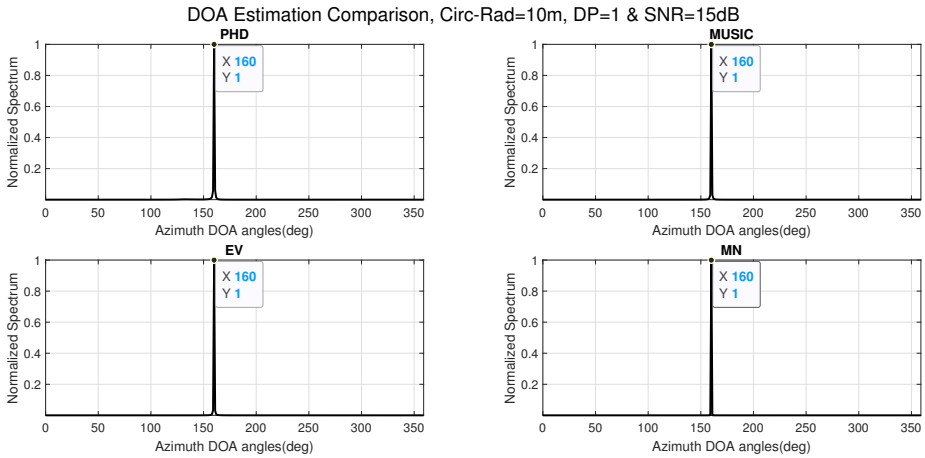


Figure 5.12: DOA estimation results in the UCA under **SNR = 15dB** with d=1 at $[160°]$.

d=2 case

The DOA estimation results for d=2 case are obtained with the same system parameters as in d=1 case and these are given under four SNR levels as in Figure 5.13 to Figure 5.16.

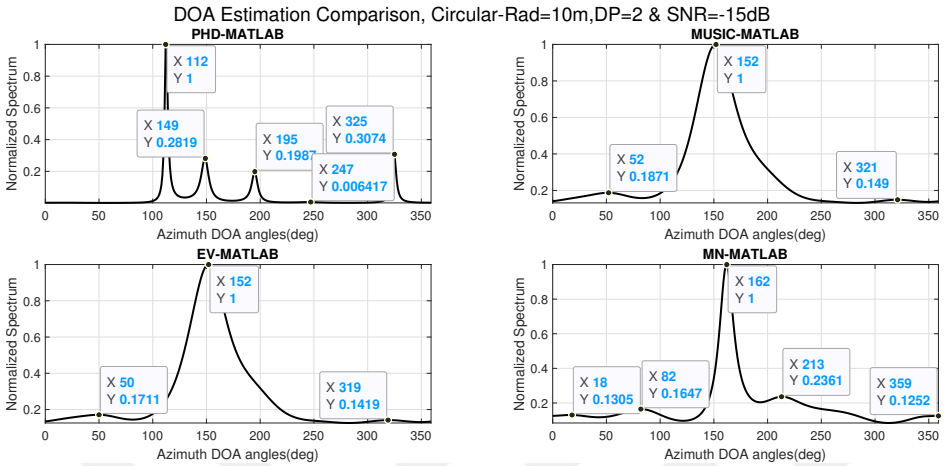


Figure 5.13: DOA estimation results in the UCA under **SNR = -15dB** with d=2 at $[150°]$ and $[200°]$.

As seen from the Figure 5.13, all methods fail to resolve two emitters and in each of them, several spurious peaks are observed. Despite this, the major peak in all the methods except PHD is obtained generally close (max $10°$ away) to one of the correct angles, which is $[150°]$.

When the SNR level is increased to -5dB, as in Figure 5.14, most of spurious peaks in the methods disappear and the max. estimation error drops to $2°$.

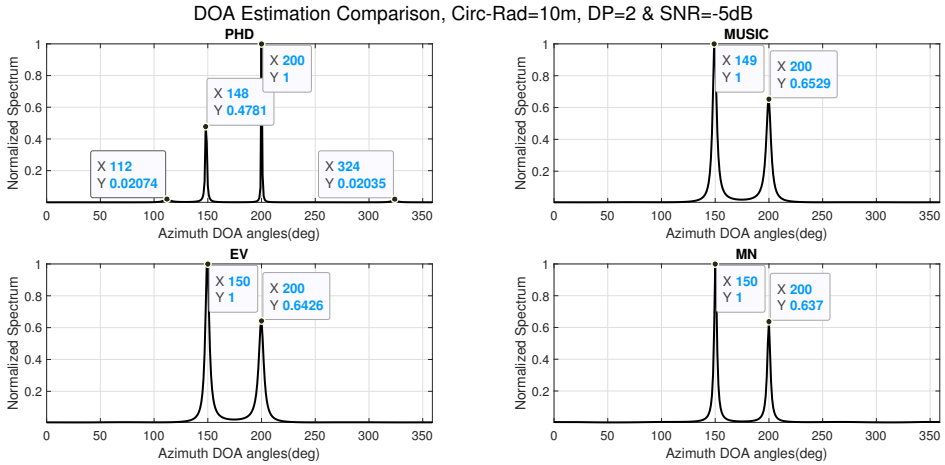


Figure 5.14: DOA estimation results in the UCA under **SNR = -5dB** with d=2 at $[150°]$ and $[200°]$.

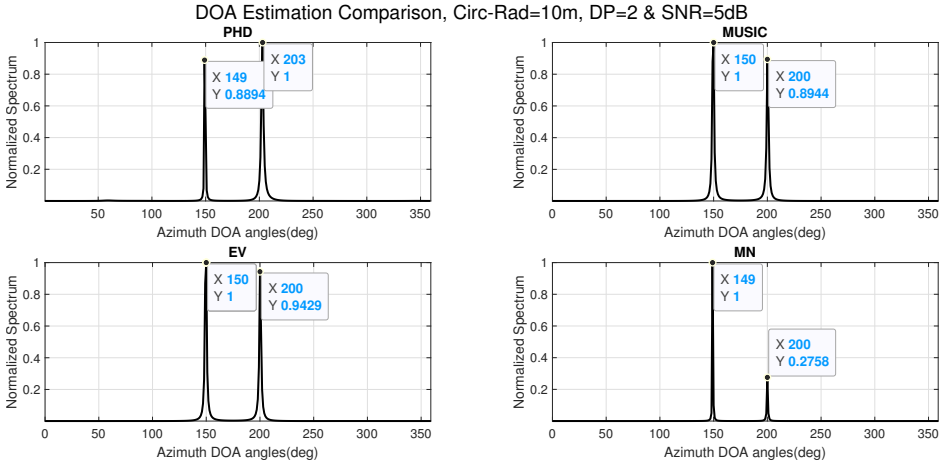


Figure 5.15: DOA estimation results in the UCA under **SNR = 5dB** with d=2 at $[150°]$ and $[200°]$.

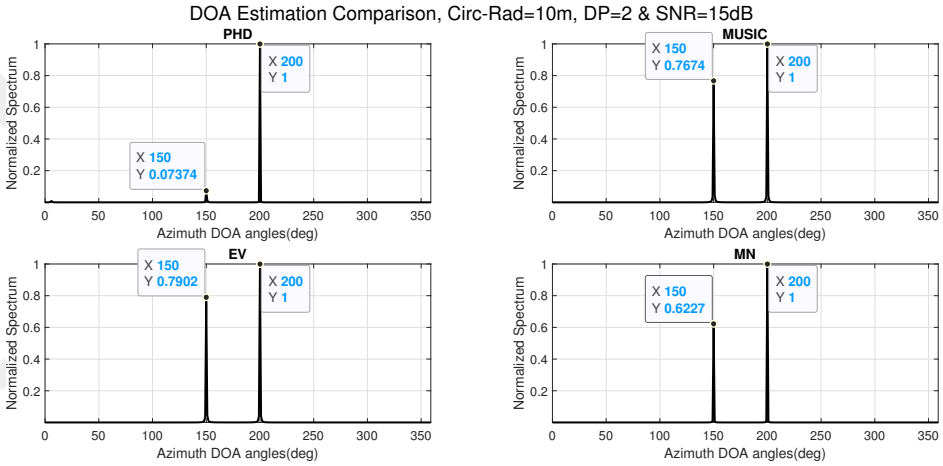


Figure 5.16: DOA estimation results in the UCA under **SNR = 15dB** with d=2 at $[150°]$ and $[200°]$.

As in Figure 5.11 and Figure 5.12, under SNR = 5dB and SNR = 15dB cases, MUSIC and EV perform without any DOA error and two others (PHD and MN) make relatively less error compared to previous SNR levels.

5.2 Numerical and Performance-based Analyses

After observing the effects of different parameters (geometry, aperture and SNR) on the DOA estimation performance of our DOA methods (PHD, MUSIC, EV, and MN) and evaluating them from the theoretical point of view in Section 5.1, the numerical and performance-based experiments for the methods are provided in this section.

5.2.1 Test Scenario

In order to validate and measure the performance of C/C++ and CUDA implementations, a specific DOA test scenario was designed and the corresponding data was generated using the synthetic signal generator (see Section 4.2) in MATLAB. This scenario was used in almost all numerical and performance-based experiments. This scenario is visually represented as follows.

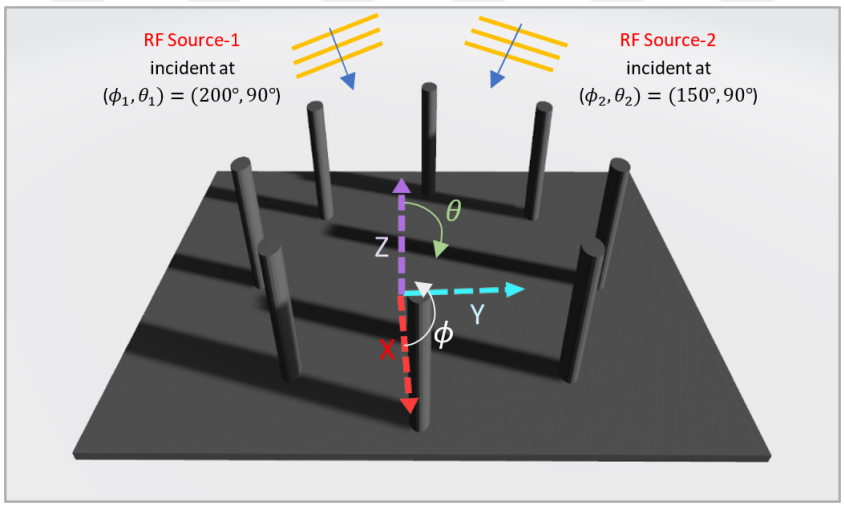


Figure 5.17: A virtual test scenario having two uncorrelated RF sources.

As can be seen from the Figure 5.17, this scenario assumes that there are two uncorrelated RF sources whose EM waves propagate only in their direct paths (i.e., no multipath) and their wavefronts impinge on the passive antenna array at $[90°]$ in elevation (perfect ground waves assumption). All the other assumptions (e.g., narrowband, far-field, etc.) mentioned in Section 2.3.2 are also valid in this scenario. The antenna array here was chosen as UCA (uniform circular array) and it consists of 8 identical monopole antennas, each of which has an omnidirectional radiation pattern, theoretically. All the details regarding to the test scenario are tabulated in Table 5.1.

Table 5.1: The detailed description of the test scenario parameters.

Antenna Array Configuration	**Array Type:**	Uniform Circular Array (UCA)
	Antenna Type:	Monopole (vertical polarization)
	Number of Antennas:	8
	Array aperture:	UCA radius = 10 m
Test Signal Properties	**Path Number(s):**	2 direct-path & no multipath
	Expected DOA Angles:	$[200°, 90°]$ & $[150°, 90°]$
	RF carrier frequency (s):	15 MHz (same for all)
	Synthetic Noise Type:	AWGN
	SNR level:	15 dB
Signal Data Properties	**Sampling Frequency:**	3e07 Hz
	Data Frame:	1000 samples (single packet)

5.2.2 General Numerical Validation Analysis

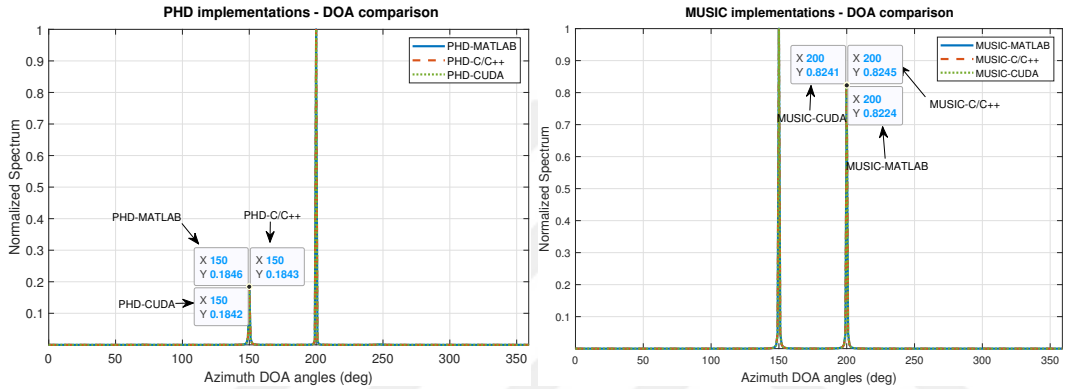
In order to check the numerical validity of C/C++ implementations of DOA methods in single precision (real/complex-float32), the pseudospectrum values corresponding to the angle scan pairs were used as percent error measurement data. Considering the test scenario given in Section 5.2.1, for the pseudospectrum scan ranges, azimuth=[0:1:359](deg) and constant elevation=[90](deg) were chosen. Thus, 360 real values were obtained for each implementation of the methods. Then, the values obtained from C/C++ and CUDA implementations were compared against the ones obtained from the ground-truth MATLAB implementations. In this analysis, for measuring the numerical accuracy in general, the percent error metric was adopted and it was computed by Equation 5.1.

$$e_{percent} = \frac{100}{N} \sum_{i=1}^{N} \frac{|y_{meas,i} - y_{gndTruth,i}|}{|y_{gndTruth,i}|} \tag{5.1}$$

The percent error results obtained from this analysis for C/C++ and CUDA implementations against MATLAB were tabulated in Table 5.2. According to this table, the maximum percent error of the single precision(float32) implementations is 0.02427%. Since the MATLAB implementations are based on double precision (float64), this error is considered acceptable. However, in order to verify that there is no DOA estimation discrepancy between the implementations, their results were compared for each method separately as in Figure 5.18 and in Figure 5.19.

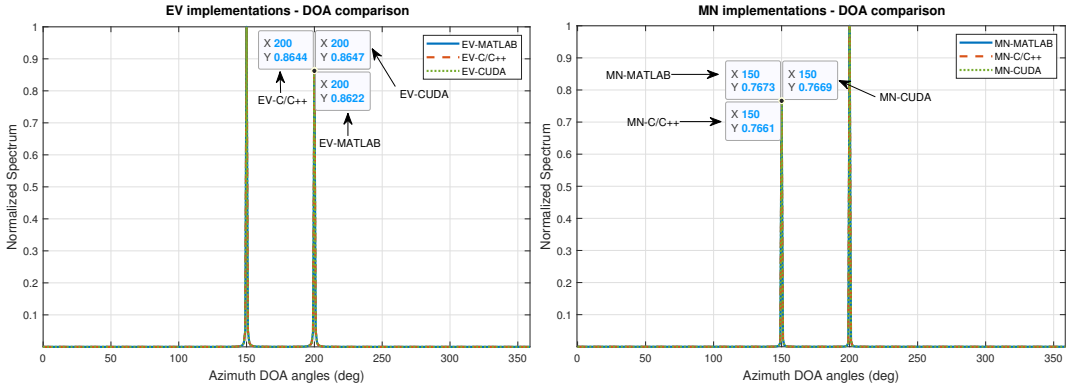
Table 5.2: Numerical percent (%) error comparison.

DOA Methods	Ground Truth	Percent (%) Error	
		C/C++ (float32)	CUDA (float32)
PHD	MATLAB (float64)	0.016952	0.024270
MUSIC		0.001229	0.001134
EV		0.001720	0.002010
MN		0.000850	0.000403



(a) (b)

Figure 5.18: DOA estimation comparison of (a) PHD and (b) MUSIC implementations.



(a) (b)

Figure 5.19: DOA estimation comparison of (a) EV and (b) MN implementations.

5.2.3 Numerical and Performance Analysis for SVD Computation

In the experiments to be mentioned here, to be able to determine which resource (CPU or GPU) and the implementation environment (C/C++ or CUDA) are more suitable for the SVD (singular value decomposition) computation, some analyses were realized in terms of numerical accuracy and computational performance.

To investigate the availability of an efficient GPU-based SVD computation, a CUDA test code was prepared by adapting from a code exampled provided officially by NVIDIA© and this was used to evaluate CUSOLVER – "gesvdj" method [52]. Similar test environments were also prepared in C/C++ for Eigen-JacobiSVD method [51] and in MATLAB for its svd method [55].

As mentioned in the test scenario (see Section 5.2.1), we are dealing with 8 sensors in our experiments and this results in an SVD of 8x8 $\widehat{R}$ matrix. In order to evaluate the numerical accuracy of all possible SVD implementations, residual error was used as metric and it was measured for each by using an 8x8 complex-valued test matrix. The residual error in our analyses for SVD is basically computed by the Equation 5.2, where $\mathbf{A}$ represent the input matrix and $\mathbf{S}$, $\mathbf{U}$ and $\mathbf{V}$ are matrices including singular values, left and right singular vectors, respectively.

$$residualErr = \left|\mathbf{A} - \mathbf{U}\,\mathbf{S}\,\mathbf{V}^{\mathrm{H}}\right| \tag{5.2}$$

Table 5.3: SVD methods - residual error comparison for an 8x8 matrix.

	MATLAB	**Eigen**	**cuSOLVER**
Res. Error:	1.075e-14	1.882e-05	6.391e-08

As can be seen from Table 5.3, (ground-truth) MATLAB-svd has the lowest residual error since it computes in double precision (float64) and single-precision alternatives in Eigen (C/C++) and cuSOLVER (CUDA) have both reasonable residual errors.

As can also be seen from Table 5.4, the MATLAB-svd outperforms the Eigen JacobiSVD in all matrix sizes. This is probably due to the fact that the MATLAB-svd is based on a LAPACK routine that computes SVD by QR decomposition with initial householder reflection and this approach theoretically requires less computations (hence faster) [56]. For small matrices (as in our case), the Eigen-JacobiSVD method performs significantly better than the cuSOLVER-gesvdj method. Even if the cuSOLVER-based SVD is computed on the GPU, due to the extensive amount of preparation steps (configurations, query workspace, etc.) and memory transfers for using the solver, its performance dramatically decreases.

By considering all numerical and performance-based analyses mentioned above, the resource and the implementation environment for all SVD computations were selected as CPU and C/C++ using Eigen-JacobiSVD method, respectively.

Table 5.4: SVD methods - performance comparison (msec).

Matrix Size:	MATLAB SVD:	Eigen JacobiSVD:	cuSOLVER gesvdj:
8x8	0.018	0.112	390.279
64x64	0.836	29.910	386.242
512x512	64.283	18651.000	N/A
1024x1024	54872.077	183033.000	N/A

5.2.4 General Performance Evaluation

Since all DOA methods share a common code structure as mentioned in Algorithm 1, it is also natural for them to have similar computational performances as given in Appendix A. Therefore, in order to comparatively evaluate the initial code performances in MATLAB, C/C++ and CUDA, out of four alternative DOA methods, the MUSIC method was chosen and used as a basis method in our performance experiments. Without any specific performance optimization, the initial performance of the MUSIC CUDA implementation was compared against serial MATLAB and serial/multi-threaded C/C++ implementations. As stated in the hotspot analysis of MUSIC C/C++ code (see Figure 4.1), the spectral search part dominates the overall execution time and this part is actually controlled by the DOA scan range (L). Starting from this fact, to observe the performance trend of the implementations, the comparison was repeated for different DOA scan ranges and the results were tabulated as in Table 5.5. In order to evaluate the performance trend for C/C++ and CUDA implementations, a performance bar chart was also presented in Figure 5.20.

Table 5.5: General performance comparison for different ranges(msec) for MUSIC.

Scan Range	MATLAB	C/C++ (serial)	C/C++ (multi)	CUDA
360 x 1	6.4	0.486	0.408	0.507
360 x 30	135.2	3.106	1.242	0.658
360 x 60	269.9	5.612	2.111	0.823
360 x 90	401.9	8.318	2.862	0.966

In this performance analysis, the time required for preparing array manifold was considered as start-up overhead and excluded from all implementations since in a con-

tinuously running stationary DOA system, this is actually created once and reused (lookup table approach). The MATLAB implementation was executed with a single thread (no explicit parallelization) and in the multi-threaded C/C++ implementation of MUSIC, 4 OpenMP threads were utilized.

As can be seen from Table 5.5 or Figure 5.20, MUSIC CUDA code generally performs better than the other ones and the performance differences between them get more pronounced as the scan range takes larger value towards (360 x 90). The obtained results show that initial CUDA implementation provides up to **2.96x** speedup compared to the multi-threaded baseline C/C++ counterpart.

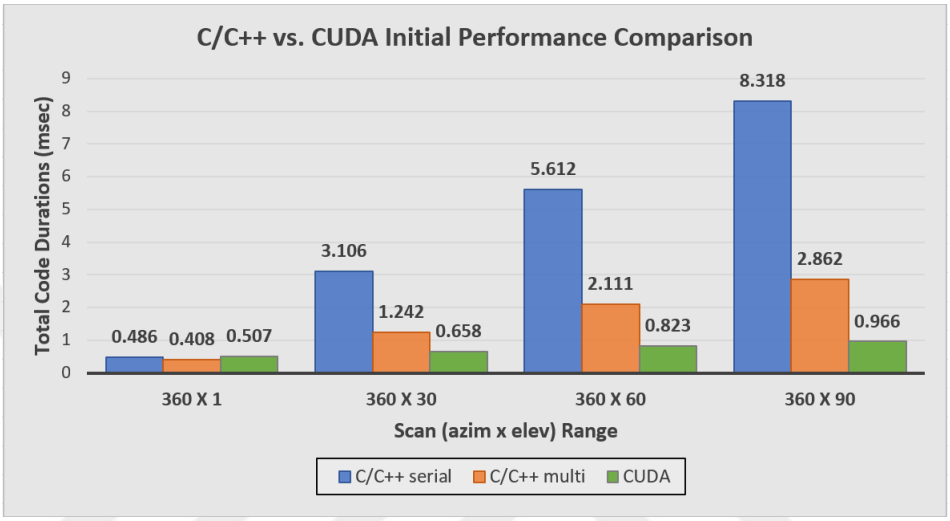


Figure 5.20: C/C++ (serial/multi-threaded) vs. CUDA initial performance comparison (msec).

5.2.5 CUDA Code Performance Optimizations

After determining the initial performance of the MUSIC CUDA implementation, in order to improve the code performance, some further CUDA optimizations were investigated in line with the CUDA design cycle (Figure 4.4).

The CUDA code optimization procedure was initiated by considering that GPU-parallelized part (see Figure 4.5) of our noise subspace-based DOA methods is governed by a single kernel function and it does not include intensive mathematical operations. This fact is due to the nature of the methods and it results in having the corresponding GPU threads to be lightweight. Taking these into account, the optimizations under our consideration were determined as ones based on CUDA-related memory access instead of the strategies like concurrent multi-kernel operations, overlapped kernel & data transfer or other low-level parallelism techniques.

All the optimization attempts are given with details in Table 5.6 and each optimization step mentioned here was incrementally applied. The performance measurements related to these optimization steps were realized for the MUSIC CUDA code in the DOA pair range where azimuth=[0:1:359](deg) and elevation=[1:1:90](deg) were scanned. In this case, the measurements were done for kernel and device-to-host

(D2H) duration as given in Table 5.7.

Table 5.6: Details on the CUDA optimization steps.

Steps:	**Optimization Details:**
1	The array C_d with complex-float entries was moved from global to local memory for any possibility of performance improvement.
2	The pointer A_table_d pointing the lookup table array containing all steering vectors was declared with two additional qualifiers, namely "**const**" and "**restrict**".
3	Host arrays (C_h, A_table_h and P_table_h) which have a direct relationship with the GPU-side were allocated in the pinned (page-locked) host memory instead of pageable host memory.

Table 5.7: CUDA code optimization steps and corresponding kernel/memcpy durations (msec).

	Optim Steps:	1	2	3
Measured part (msec)	**Initial Version**	**C: global->local**	**const & restrict**	**Pinned mem-alloc**
Kernel:	0.129	0.128	0.055	0.054
D2H memcpy:	0.098	0.098	0.097	0.072

- **Optim Step-1:** Changing the memory place of the array C from global to local memory did not result in any positive performance change. The local memory (lmem) is actually an off-chip memory (on the GDRAM) dedicated privately to each thread and its name does not indicate the location that it resides. Therefore, the cost for the access to this is generally same as the global memory [47]. It is generally used for spilling by the CUDA program when too much register space is consumed. Apart from all these, lmem stores can also be cached in L1 (Pascal and Kepler) depending on the compiler behavior for addressing and with this, some memory traffic may be avoided [57]. Therefore, this optimization step was just done experimentally without a significant expectation.
- **Optim Step-2:** With the additional qualifiers to the pointer A_table_d, the kernel performance was improved by "**2.4x**" reducing the kernel execution time from 0.128 msec to 0.055 msec. These qualifiers actually enforce the compiler to use the read-only cache (48 KB per SM) in texture-cache memory space

without allowing pointers to alias one another [43]. The read-only cache can also be equivalently activated by "**ldg()**" intrinsic function [58]. In our case, since steering vector parts of the data pointed by A_table_d are just read by the threads and there is no write operation on this array, the read-only cache usage was suitable.

- **Optim Step-3:** Since C_h is quite small and A_table-related durations were accepted as overhead, only copying data from device to host (D2H) concerning P_table_h was measured in this step. As seen, D2H memcpy operation performance was improved by "**1.35x**" reducing the kernel execution time from 0.128 msec to 0.055 msec. The pinned memory allocation basically prevents the operating system (OS) from paging the corresponding memory out to disk and hence, it makes sure that the data resides in the physical memory (RAM). In our case, this situation basically enables the GPU to use direct memory access (DMA) for H2D and D2H operations by knowing the physical address [59].

5.2.6 CUDA Code Performance Benchmarking on Different Platforms

In order to observe the effect of the system specifications on the optimized CUDA code performance and its speedup compared to baseline C/C++ implementation, a performance benchmarking analysis was realized using the MUSIC CUDA code. Within the scope of these analyses, the performance measurements were done on three different system configurations whose specifications are given in Table 5.8 briefly and in Appendix B in detail.

Table 5.8: Technical specs comparison for different configurations used in the benchmarking.

Specs	**Config-1**	**Config-2**	**Config-3**
CPU	Intel i7-9750H	Intel i7-7700HQ	Intel Xeon W-2255
CPU clock	2.6 GHz	2.8 GHz	3.7 GHz
CPU cores	6 cores	4 cores	10 cores
GPU device	GTX 1660 Ti	GTX 1050	RTX 2080 Ti
Architecture	Turing	Pascal	Turing
CUDA cores	1536	640	4352
GPU clock	1335 MHz	1493 MHz	1545 MHz
Memory clock	6001 MHz	3504 MHz	7000 MHz
Mem bus width	128-bit	192-bit	352-bit

Table 5.9: MUSIC CUDA code performance comparison (msec) and speedup against baseline multi-threaded MUSIC C/C++ code on different system configurations.

	Config-1		**Config-2**		**Config-3**	
Scan Range	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup
360 x 1	0.438	0.93x	0.677	0.60x	0.526	0.77x
360 x 30	0.647	1.92x	1.017	1.22x	0.649	1.92x
360 x 60	0.812	2.60x	1.257	1.68x	0.663	3.19x
360 x 90	0.924	3.10x	1.447	1.98x	0.880	**3.25x**

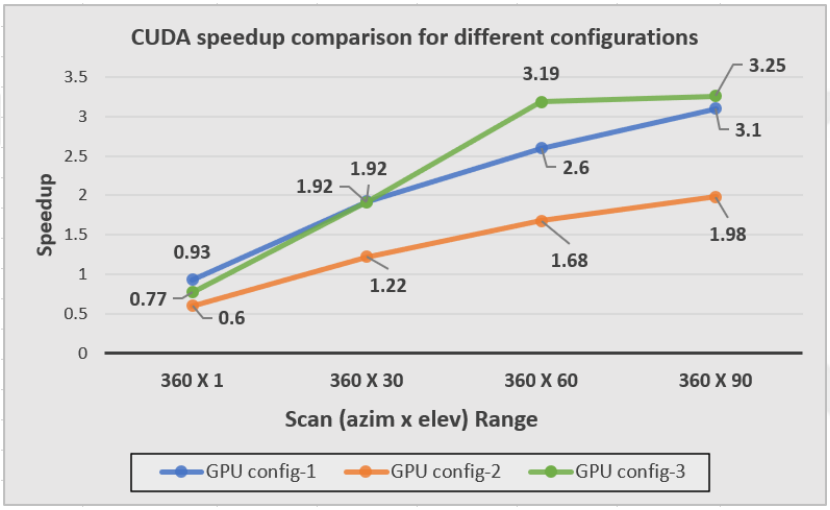


Figure 5.21: MUSIC CUDA code speedup against baseline C/C++ code for different configurations.

As shown in Table 5.9, the MUSIC CUDA code on Config-1 performs approximately $1.5\times$ faster than the one on Config-2 for the largest scan range in our analysis (360 x 90). When the GPU-related technical specifications are especially compared with each other, the difference between their performances seems to be reasonable.

Even if the MUSIC CUDA code performances on Config-1 and Config-3 are comparable, the performance on Config-3 exceeds the one on Config-1 starting from the scan range (360 x 30).

As can be seen from Figure 5.21, even if the technical specifications seem not to be so significant (either Config-1, Config-2 or Config-3) when we evaluate the speedup at low scan ranges like (360 x 1), the choice of the system becomes more critical as the speedup is evaluated for higher scan ranges towards (360 x 90). The experiment results show that **up to 3.25x speedup** can be achieved by the CUDA implementation on an available configuration against the multi-threaded C/C++ implementation. When we consider a real-time DOA estimation system, the benchmarking analysis similar to the one here is very crucial for determining the conditions to satisfy an acceptable time performance and a high DOA scan range at the same time.

CHAPTER 6

CONCLUSION

In this thesis, four noise subspace-based DOA methods (PHD, MUSIC, EV and MN) have been implemented in MATLAB, C/C++ and CUDA. We have studied the algorithms from a theoretical perspective and evaluated them in terms of numerical accuracy and computational performance on CPU and GPU.

Within the scope of theoretical analyses, DOA estimation accuracy was observed via a set of controlled experiments under different system parameters such as array geometry, aperture size and SNR-level. Conducting these experiments has contributed to the determination of typical system parameter values in order to satisfy an expected average DOA estimation performance. These nominal system values have helped us later to determine which system parameter value dominates in the complexity analysis and hence which computation step(s) in our algorithms is/are hotspot(s) for us.

In order to determine the parallelization structure of our DOA estimation methods on CPU and GPU, some prior studies like computational complexity analysis and hotspot analysis have been realized. These studies basically helped us to determine which steps in our algorithms are critical in terms of execution time. Based on this information, parallel CPU and GPU implementations have been realized. After that point on, GPU-based CUDA implementations were mainly focused and at the first step, their parallel code performances were initially assessed. Afterwards, in order to improve their code performances, some GPU-based optimizations were applied and their effects were evaluated. At the last step, some performance-based benchmarking studies have been conducted on different system configurations and the suitability of our algorithms for scalability was observed. In addition to performance-based studies, some numerical validation analyses have also been conducted. In these analyses, a particular attention has been devoted to the minimization of numerical error-based effects on the DOA estimation performance by adapting most appropriate implementation approaches.

After the optimization and performance tuning steps, on an available configuration, up to $3.25\times$ speedup was consequently achieved by the MUSIC CUDA implementation against the multi-threaded MUSIC C/C++ implementation. The resultant implementations in CUDA are numerically stable and sufficiently fast ($<$ 1 msec) realizations of noise subspace-based DOA estimation methods. Therefore, from a practical point of view, they can be efficiently utilized in real-time DOA systems.

Some possible future works for this thesis are as follows:

1. In addition to the system-based parameters that were studied, the effect of the number of antenna elements and the number of snapshots on the DOA estimation performance can also be studied to understand the nature of DOA estimation in a better way.

2. For hemispherical or full-spherical 3D case with a sufficient angular resolution, instead of scanning azimuth and elevation in equal angle steps, it is possible to adapt much more efficient sampling schemes like HEALPix which provides equal-area iso-latitude sphere pixelation [60].

3. Finding the peaks of the pseudo-spectrum obtained at the end of the kernel execution can be realized on GPU instead of copying the data back to CPU. This can be done by an appropriate approach like max-reduction using shared memory.

4. In order to reduce the computational cost of the algorithms, instead of following pseudo-spectra approach, spatial entropy-based approach used in the acoustics [61] can be adapted to our RF-based scenario.

5. The SVD operation in the algorithms could be implemented using alternative numerical libraries, such as KBLAS [62] or Intel MKL [63].

6. To make our implementations more suitable for real applications, multipath propagation case can also be taken into consideration and the algorithms can be adapted for identifying path characteristics (either direct-path or multipath) with the help of some suitable methods in the literature (forward/backward spatial smoothing for ULA case, etc.).

7. Instead of studying narrowband DOA estimation methods considering the signals at a single frequency, broadband signals can be considered by adopting appropriate changes in the code structure and the number of frequencies can be utilized as another dimension in the overall parallelization structure.

REFERENCES

[1] Z. Chen, G. Gokeda, and Y. Yu, *Introduction to Direction-of-arrival Estimation*. Artech House, 2010.

[2] T. E. Tuncer and B. Friedlander, *Classical and modern direction-of-arrival estimation*. Academic Press, 2009.

[3] C.-S. Park and D.-Y. Kim, "The fast correlative interferometer direction finder using i/q demodulator," in *2006 Asia-Pacific Conference on Communications*, pp. 1–5, IEEE, 2006.

[4] E. Bellini and A. Tosi, "System of directed wireless telegraphy.," Dec. 21 1909. US Patent 943,960.

[5] D. N. Travers and S. M. Hixon, "Abstracts of the available literature on radio direction finding 1899-1965," tech. rep., Southwest Research Institute San Antonio TX, 1966.

[6] H. Krim and M. Viberg, "Two decades of array signal processing research: the parametric approach," *IEEE signal processing magazine*, vol. 13, no. 4, pp. 67–94, 1996.

[7] M. W. Majid, T. E. Schmuland, and M. M. Jamali, "Parallel implementation of the wideband doa algorithm on single core, multicore, gpu and ibm cell be processor," *Science Journal of Circuits, Systems and Signal Processing*, vol. 2, no. 2, p. 29, 2014.

[8] M. Kim, K. Ichige, and H. Arai, "Real-time smart antenna system incorporating fpga-based fast doa estimator," in *IEEE 60th Vehicular Technology Conference, 2004. VTC2004-Fall. 2004*, vol. 1, pp. 160–164, IEEE, 2004.

[9] Z. Zou, W. Hongyuan, and Y. Guowen, "An improved music algorithm implemented with high-speed parallel optimization for fpga," in *2006 7th International Symposium on Antennas, Propagation & EM Theory*, pp. 1–4, IEEE, 2006.

[10] M. W. Majid, G. Mirzaei, and M. M. Jamali, "Parallel wavelet transform based wideband direction-of-arrival (doa) on multicore/gpu," in *IEEE International Conference on Electro-Information Technology, EIT 2013*, pp. 1–4, IEEE, 2013.

[11] Z. Lu, L. Zhang, J. Zhang, and J. Zhang, "Parallel optimization of broadband underwater acoustic signal music algorithm on gpu platform," in *2017 4th International Conference on Systems and Informatics (ICSAI)*, pp. 704–708, IEEE, 2017.

[12] R. Farber, *CUDA application design and development*. Elsevier, 2011.

[13] J. I. Buskenes, J. P. Åsen, C.-I. C. Nilsen, and A. Austeng, "An optimized gpu implementation of the mvdr beamformer for active sonar imaging," *IEEE Journal of Oceanic Engineering*, vol. 40, no. 2, pp. 441–451, 2014.

[14] C. Sarofeen and P. Gillett, "A high performance parallel and heterogeneous approach to narrowband beamforming," *IEEE transactions on parallel and distributed systems*, vol. 27, no. 8, pp. 2196–2207, 2015.

[15] N. Kozic, I. Pokrajac, and P. Okiljevic, "An approach of using general purpose graphics processing units in modern systems for electronic warfare," in *International Conference on Aerospace Sciences and Aviation Technology*, vol. 17, pp. 1–7, The Military Technical College, 2017.

[16] H. Eray and A. Temizel, "Performance analysis of noise subspace-based narrowband direction-of-arrival (doa) estimation algorithms on cpu and gpu," *6th High Performance Computing Conference (BAŞARIM 2020)*.

[17] "Loran–the new radio navigator," *Hydrographic Review*, no. 1400, pp. 9–30, 1946.

[18] "Cospas-sarsat system." `http://www.cospas-sarsat.int/`.

[19] J. L. Volakis, *Antenna engineering handbook*. McGraw-Hill Education, 2007.

[20] J. D. Kraus, M. Tiuri, A. V. Räisänen, and T. D. Carr, *Radio astronomy*, vol. 69. Cygnus-Quasar Books Powell, Ohio, 1986.

[21] F. Adcock, "Improvement in means for determining the direction of a distant source of electro-magnetic radiation," UK 130,490 Aug. 7, 1919.

[22] I. Pellejero, "Adcock/watson-watt radio direction finding." `https://ipellejero.es/tecnico/adcock/english.php`.

[23] R. Hammerle, "Factors limiting the accuracy of doppler and adcock direction finding systems," in *IEE Colloquium on Passive Direction Finding*, pp. 3/1–313, 1989.

[24] J. B. Alex, "A comparison of the watson-watt and pseudo-doppler df techniques–rf products web note wn-004, april, 1999."

[25] R. Chellappa and S. Theodoridis, *Academic Press Library in Signal Processing, Volume 7: Array, Radar and Communications Engineering*. Academic Press, 2017.

[26] J. Capon, "High-resolution frequency-wavenumber spectrum analysis," *Proceedings of the IEEE*, vol. 57, pp. 1408–1418, Aug 1969.

[27] V. F. Pisarenko, "The retrieval of harmonics from a covariance function," *Geophysical Journal International*, vol. 33, no. 3, pp. 347–366, 1973.

[28] B. Ottersten, M. Viberg, P. Stoica, and A. Nehorai, "Exact and large sample maximum likelihood techniques for parameter estimation and detection in array processing," in *Radar array processing*, pp. 99–151, Springer, 1993.

[29] S. Häfner, M. Käske, and R. Thomä, "On calibration and direction finding with uniform circular arrays," *International Journal of Antennas and Propagation*, vol. 2019, 2019.

[30] P. S. Naidu, *Sensor array signal processing*. CRC press, 2009.

[31] C. A. Balanis, *Antenna theory: analysis and design*. John wiley & sons, 2016.

[32] V. MADISETTI, "Wireless, networking, radar, sensor array processing, and nonlinear signal processing (the digital signal processing handbook," 2009.

[33] S. Marple, *Digital spectral analysis with applications*. Prentice-Hall, Inc, 1987.

[34] M. H. Hayes, *Statistical digital signal processing and modeling*. John Wiley & Sons, 2009.

[35] R. O. Schmidt, "Multiple emitter location and signal parameter estimation," *IEEE Transactions on Antennas and Propagation*, vol. 34, pp. 276–280, 1986.

[36] D. Johnson and S. DeGraaf, "Improving the resolution of bearing in passive sonar arrays by eigenvalue analysis," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 30, no. 4, pp. 638–647, 1982.

[37] R. Kumaresan and D. W. Tufts, "Estimating the angles of arrival of multiple plane waves," *IEEE Transactions on Aerospace and Electronic Systems*, no. 1, pp. 134–139, 1983.

[38] M. Kaveh and A. Barabell, "The statistical performance of the music and the minimum-norm algorithms in resolving plane waves in noise," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 34, no. 2, pp. 331–341, 1986.

[39] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, vol. 10. MIT press, 2008.

[40] "Openmp." `https://www.openmp.org/`.

[41] R. Van der Pas, E. Stotzer, and C. Terboven, *Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD*. MIT Press, 2017.

[42] "Cuda c++ programming guide." `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[43] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[44] "Cpu and gpu databases - techpowerup." `https://www.techpowerup.com/`.

[45] "Performance comparison of cpus and gpus through the last decade or so." `https://github.com/mgalloy/cpu-vs-gpu`.

[46] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[47] NVIDIA, "Best practices guide :: Cuda toolkit documentation." https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[48] NVIDIA, "Nvcc :: Cuda toolkit documentation." https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[49] M. Kim, K. Ichige, and H. Arai, "Implementation of fpga based fast doa estimator using unitary music algorithm [cellular wireless base station applications]," in *2003 IEEE 58th Vehicular Technology Conference. VTC 2003-Fall (IEEE Cat. No. 03CH37484)*, vol. 1, pp. 213–217, IEEE, 2003.

[50] F. Dellaert, "Singular value and eigenvalue decompositions." https://www.cc.gatech.edu/~dellaert/pubs/svd-note.pdf, May 2008.

[51] I.-T. Family, "Eigen c++ template library." http://eigen.tuxfamily.org/.

[52] N. Corporation, "cusolver." https://docs.nvidia.com/cuda/cusolver/index.html.

[53] H. L. Van Trees, *Optimum array processing: Part IV of detection, estimation, and modulation theory*. John Wiley & Sons, 2004.

[54] S. Chandran, *Advances in direction-of-arrival estimation*. Artech House Boston, 2006.

[55] MathWorks, "Matlab-svd." https://www.mathworks.com/help/matlab/ref/double.svd.html.

[56] "Eigen: Benchmark of dense decompositions." https://eigen.tuxfamily.org/dox/group__DenseDecompositionBenchmark.html.

[57] P. Micikevicius, "Cuda register spilling." https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf.

[58] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[59] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 2010.

[60] "Healpix." https://healpix.sourceforge.io/.

[61] O. Olgun and H. Hacihabiboglu, "Localization of multiple sources in the spherical harmonic domain with hierarchical grid refinement and eb-music," in *2018 16th International Workshop on Acoustic Signal Enhancement (IWAENC)*, pp. 101–105, 2018.

[62] K. E. C. R. Center, "Kblas-gpu." https://github.com/ecrc/kblas-gpu.

[63] Intel, "oneapi-mkl." https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html.

APPENDIX A

ALL INITIAL PERFORMANCE RESULTS

Initial performance results for MATLAB and C/C++ implementations of all four noise subspace-based DOA methods (PHD, MUSIC, EV, and MN) are given in Table A.1.

Table A.1: Initial performance results for MATLAB vs. C/C++ implementations

Scan Range (Az & Elev):	Total # of cost values:	Total Code Durations:							
		PHD-MATLAB	PHD-C/C++	MUSIC-MATLAB	MUSIC-C/C++	EV-MATLAB	EV-C/C++	MN-MATLAB	MN-C/C++
[0:1:359] & [90:1:90]	360*1	5.7 msec	T3: 0.402 msec T4: 0.430 msec	6.4 msec	T3: 0.339 msec T4: 0.408 msec	6.6 msec	T3: 0.448 msec T4: 0.371 msec	9.3 msec	T3: 0.379 msec T4: 0.413 msec
[0:1:359] & [61:1:90]	360*30	137.2 msec	T3: 1.311 msec T4: 1.230 msec	135.2 msec	T3: 1.285 msec T4: 1.242 msec	136.8 msec	T3: 1.389 usec T4: 1.180 msec	136.6 msec	T3: 1.394 msec T4: 1.194 sec
[0:1:359] & [31:1:90]	360*60	277.6 msec	T3: 2.369 msec T4: 2.130 msec	269.9 msec	T3: 2.297 msec T4: 2.111 msec	271.8 msec	T3: 2.452 msec T4: 2.015 msec	268.8 msec	T3: 2.419 msec T4: 2.079 msec
[0:1:359] & [01:1:90]	360*90	404.2 msec	T3: 3.397 msec T4: 2.905 msec	401.9 msec	T3: 3.423 msec T4: 2.862 msec	409.4 msec	T3: 3.481 msec T4: 2.974 msec	402.1 msec	T3: 3.494 msec T4: 2.979 msec

Explanations:

- All MATLAB codes work in single thread and **no parfor** (parallel for) option or any additional acceleration option is preferred.
- **T3 & T4** notations given in the table represent the duration measurement with **3 & 4 active OpenMP threads**.

APPENDIX B

ALL SYSTEM SPECIFICATIONS

Table B.1: Detailed system specs comparison for Config-1 and Config-2.

sysinfo / devQuery	Config-1	Config-2
Processor Freq	Intel i7-9750H 2.6 GHz	Intel i7-7700HQ 2.8 GHz
Number of CPU cores	6 Physical (12 Logical)	4 Physical (8 Logical)
Physical Memory	16 GB	16 GB
Page File Space	2.88 GB	13.6 GB
GPU-device	GeForce GTX 1660 Ti	GeForce GTX 1050
Installed CUDA Driver	10.2	10.1
CUDA Capability	7.5	6.1
Number of SMs	24	5
Number of CUDA cores	1536	640
GPU memory	6144 MB (6 GB)	4096 MB (4 GB)
GPU Max Clock Rate	1335 MHz	1493 MHz
Memory Clock Rate	6001 MHz	3504 MHz
Memory Bus Width	192-bit	128-bit
Constant Memory	65536 bytes	65536 bytes
Shared memory/block	49152 bytes	49152 bytes
Registers/block	65536	65536
Warp Size	32	32
Max threads/SM	1024	2048
Max threads/block	1024	1024
Concurrent execution	Yes with 6 copy engines	Yes with 5 copy engines

Table B.2: Detailed system specs for Config-3.

sysinfo / devQuery	Config-3
Processor Freq	Intel Xeon W-2255 3.7 GHz
Number of CPU cores	10 Physical (20 Logical)
Physical Memory	32 GB
Page File Space	2.38 GB
GPU-device	GeForce RTX 2080 Ti
Installed CUDA Driver	11.0
CUDA Capability	7.5
Number of SMs	68
Number of CUDA cores	4352
GPU memory	11264 MB (11 GB)
GPU Max Clock Rate	1545 MHz
Memory Clock Rate	7000 MHz
Memory Bus Width	352-bit
Constant Memory	65536 bytes
Shared memory/block	49152 bytes
Registers/block	65536
Warp Size	32
Max threads/SM	1024
Max threads/block	1024
Concurrent execution	Yes with 3 copy engines