GPU Parallelization of the MVDR Beamforming Algorithm for Multiple Input Multiple Output Radar

Parallélisation sur GPU de l'algorithme de formation de faisceaux MVDR pour le radar à entrées multiples et à sorties multiples

A Thesis Submitted to the Division of Graduate Studies
of the Royal Military College of Canada
by

Gillian Frances Rideout, BEng, rmc

Captain

In Partial Fulfillment of the Requirements for the Degree of
Master of Applied Science in Electrical Engineering

September 2019

Acknowledgements

This thesis was made possible due to the support of my supervisor, Dr. Mostafa Hefnawi. His knowledge in the field of communications, beamforming, and MIMO radar provided the background necessary to carry out this research. His ability to provide solutions and motivation proved invaluable to completion of this thesis. I would also like to thank Dr. Vincent Roberge, whose instruction in parallel programming was crucial to this thesis and my understanding of Graphical Processor Units and their theory. Finally, I would like to thank the Royal Canadian Air Force Aerospace Warfare Centre, whose generous sponsorship made this thesis possible.

Abstract

Using a Multiple Input Multiple Output (MIMO) radar provides numerous advantages, such as improving the spatial resolution, the immunity to interference, the signal-to-noise ratio (SNR) and the probability of detection for targets. Also, MIMO radar is beneficial as it can scan an entire region in a few pulses, and much faster than a phased array, which requires a scanning beam throughout the entire region. However, MIMO radar requires significant signal processing, which can introduce significant latency; this can be solved by using a Graphics Processor Unit (GPU). GPUs contain thousands of cores and can execute many operations in parallel. This capacity enables them to execute tasks significantly faster and more efficiently than Central Processing Units (CPUs). This thesis examines the design, development, and simulation of a parallel GPU implementation of minimum variance distortionless (MVDR) beamforming for a MIMO radar system. This algorithm was compared to a tradition CPU version. In all cases, it was found that the GPU executed the algorithm faster than the CPU version while successfully detecting all targets. This result proves that GPUs can be successfully integrated within a MIMO radar setup and would aide in decreasing the overall signal processing time required for target detection. This would be an excellent inclusion in massive MIMO radar, where hundreds of transmitters and receivers will be used.

Résumé

L'utilisation d'un radar MIMO (Multiple Input Multiple Output) offre de nombreux avantages par rapport au radar à balayage de phase; il permet l'amélioration de la résolution spatiale, de l'immunité aux interférences, du rapport signal-sur-bruit (SNR) et de la probabilité de détection des cibles. Il permet aussi d’analyser une région entière en quelques impulsions et beaucoup plus rapidement qu'un radar à balayage de phase. Cependant, le radar MIMO nécessite un traitement de signal important qui peut introduire une latence significative. Ce problème peut être résolu à l'aide d'un processeur graphique (GPU). Les GPU intègrent des milliers de cœurs et peuvent exécuter de nombreuses opérations en parallèle. Cette capacité leur permet d'exécuter des tâches bien plus rapidement et efficacement que les unités centrales (CPU). Cette thèse examine la conception, le développement et la simulation d’une implémentation parallèle sur GPU de l’algorithme à formation de faisceaux MVDR (Réponse sans distorsion à variance minimale) pour un système radar MIMO. Cet algorithme a été comparé à une version sur processeur traditionnel. Dans tous les cas, il a été constaté que GPU exécute l'algorithme plus rapidement que la version sur CPU tout en détectant avec succès toutes les cibles. Ce résultat prouve que les GPU peuvent être intégrés avec succès dans une configuration radar MIMO pour réduire le temps de traitement du signal global nécessaire à la détection de cibles. Ce serait une excellente intégration dans les radar MIMO à grande échelle, où des centaines d'émetteurs et de récepteurs seront utilisés.

Contents

List of Tables

List of Figures

List of Acronyms

ACC	Adaptive Cruise Control
ALU	Arithmetic Logic Unit
AOI	Area of Interest
CPI	Coherent Processing Interval
CPU	Central Processing Unit
CUDA	Common Uniform Developer Architecture
DOA	Direction of Arrival
FMCW	Frequency Modulated Continuous Wave
GPU	Graphical Processor Unit
GPGPU	General Purpose Graphical Processor Unit
ID	Identification
LU	Lower Upper
MSE	Mean Square Error
MIMO	Multiple Input Multiple Output
MVDR	Minimum Variance Distortionless Response
RAM	Random Access Memory
RCS	Radar Cross Section
SINR	Signal to Interference Noise Ratio
SM	Stream Multiprocessor
TDM	Time Division Multiplexing
ULA	Uniform Linear Array

1 Introduction

1.1 Background

Multiple Input Multiple Output (MIMO) radar systems is a type of phased array radar which has multiple transmit elements and multiple receive elements in a single radar system. Once these signals are returned to the system, extensive signal processing is then required to identify targets. This is usually done through a combination of hardware and software. Historically, software is run using a Central Processing Unit (CPU), whose parallel computing capability is limited to the number of CPU cores in the system. This typically means that signal processing can take a significant amount of time.

However, there is an innovative solution to this problem. Graphics Processor Units (GPUs) contain thousands of cores and can execute software programs in parallel. The number of cores available implies that a GPU can deliver high throughput on a significantly large data set (hundreds of thousands of data points, depending on the size and type of the data to be manipulated) [1]. The trick with using a GPU is to ensure that the algorithm will benefit from the effort to parallelize its operation. It is lucky then that many algorithms used within MIMO radar systems are highly parallel and computationally intensive, which makes them ideal candidates for a GPU implementation [1]. GPUs are also capable of delivering real-time throughput and can be reconfigured quickly to handle changing workloads [1].

While CPU performance has flat lined, GPU performance continues to grow exponentially as technology is developed [2].

1.2 Problem Statement

The minimum variance distortionless response (MVDR) algorithm has been thoroughly investigated for traditional phased array radars. For MIMO radar systems, however, it requires matrix inversions for each pulse that is transmitted by the MIMO radar. In a small system, the time to complete these calculations on a CPU may be acceptable. However, when a system grows to be quite large (such as in the case of massive MIMO where the transmitters and/or receivers can be in the hundreds or thousands), the required calculation time to carry out the MVDR beamforming algorithm would be enormous and unacceptable. While a MIMO radar has benefits over a phased array, it is more computationally intensive. For example, in [3], it was shown that a MIMO radar can detect a target 13.3 times faster than a phased array [3].

A survey paper completed in 2017 showed that there has been much success in using GPUs in traditional single input single output (SISO) radar systems [4]; significant computational speedups were found to have occurred. As well, several papers were found which had success integrating GPU and MIMO in communication scenarios that provided optimization or greater accuracy [1], [5]–[10]. However, there was little research to be found with respect to integrating MIMO radar and GPUs.

1.3 Thesis Statement

This thesis explores the use of GPU technology in a traditional MIMO radar. More specifically, the objective of this thesis is to develop a parallel MVDR algorithm that will be used on a simulated MIMO radar system. The developed algorithm will be compared against a traditional serial algorithm.

1.4 Methodology

Since this thesis will be simulation based, an iterative programming approach will be used to ensure validity of the obtained results.

The first phase undertaken to develop the proposed algorithm in the thesis statement is to create a simplified version of the problem presented. To do this, a uniform linear array (ULA) system will be created in MATLAB®. The ULA will be used to test the validity of the designed MVDR algorithm to ensure correct results are obtained using both serial (CPU) and parallel (GPU) implementations. Various simulations are conducted with varying targets and varying array lengths.

Finally, the last portion of this thesis conducts simulations using a traditional MIMO radar setup and the developed MVDR algorithm. In this portion, targets are defined using Radar Cross Section (RCS) and are simulated to produce radar returns. Various simulations are run with the same target and different array lengths to determine the execution time of the developed GPU-based MVDR algorithm.

1.5 Thesis Outline

Chapter 2 presents a high-level overview of MIMO radar and GPUs. It presents the definition of a MIMO radar system, and specifically discusses virtual array creation, beamforming, and MVDR beamforming. This chapter also presents a high-level overview of GPUs, their history, architecture, and how they can be used in heterogeneous computing.

Chapter 3 outlines the methodology that was used to design and program the serial and parallel version of the MVDR algorithm. It describes the initial ULA setup used to simulate the developed MVDR algorithm as well as the results that were obtained.

Chapter 4 discusses the MIMO radar setup that is created to test the MVDR algorithm. It outlines the design parameters, simulation, and provides an in-depth analysis of the results that were obtained.

Chapter 5 concludes this thesis and discusses area of future research for the use of MIMO radar and GPUs.

2 Literature Review

2.1 MIMO Radar

In a MIMO radar system, more than one antenna is capable of transmitting a signal while more than one antenna is also receiving the signal which creates different transmit-receive paths; Figure 2.1 shows a MIMO radar, with *M* transmitters and *N* receivers [11]. The data collected from the multiple transmitted/received signals is processed together [12].

The operation of MIMO radar falls under one of two main configurations. In the first, the transmit and receive array elements are widely spaced, which will provide independent scattering responses for each antenna pairing. This is sometimes referred to as statistical MIMO radar [11]. In the second configuration, the transmit and receive array elements are closely spaced, also known as coherent MIMO radar [11]. This thesis will focus on the second configuration and transmitting orthogonal independent waveforms are considered.

In a systems with *M* independent transmitters and *N* receivers, *MN* paths are produced for the return from the *k*th target [3] . The generic form of the total received signal in this configuration is the sum of returns from all targets and can be expressed as [3]

$$y_n[n] = \sum_{k=1}^{K} \alpha(\theta_k) \sum_{m=1}^{M} e^{-j\omega_c \tau_{mn}(\theta_k)} s_m[n] + w_n[n] \tag{2.1}$$

where α is the complex amplitude of the k^{th} return signal from a target located at angle θ_k, s_m are the baseband samples of the m^{th} transmitted signal, $\tau_{mn}(\theta_k)$ is the phase delay between the m^{th} transmitted element, the k^{th} target and the n^{th} receiver, ω_c is the carrier's angular frequency, *[n]* is the time index, and $w_n[n]$ is the additive white Gaussian noise with a covariance matrix $R_w = \sigma^2 I$ where I is the identity matrix [3].

It is important to note that unlike a phased array radar which only contains returns from its directed beam, MIMO radar systems can scan the desired area in as little as one pulse instead of scanning a beam throughout the entire region as long as the MIMO radar can detect the received paths independently through the use of orthogonal waveforms and matched filters [3]. Separating the transmit and receive paths is critical, as these signals are used to create the virtual array to be used by the MVDR beamforming algorithm [3]. However, due to this greater search area, less power is directed towards the target which results in a lower signal to noise (SNR) ratio of the target which can lead to issues with detection [3]. This can be mitigated if pulses are integrated over a longer coherent processing interval [3].

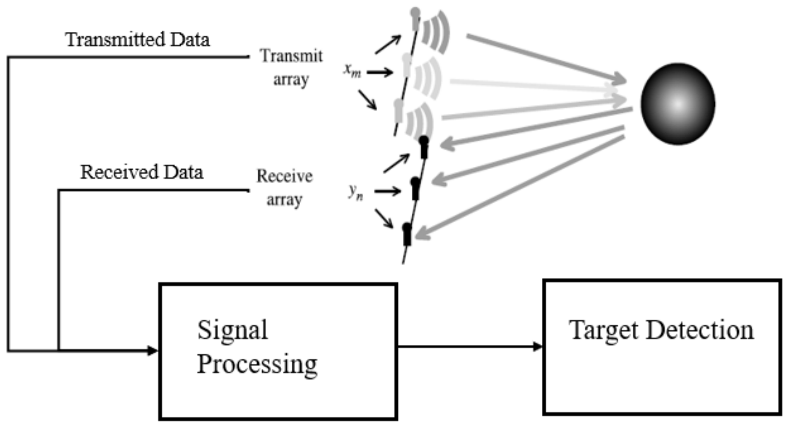


Figure 2.1 Generic MIMO radar with one target (modified from [11])

2.1.1 Orthogonal Waves

Orthogonal signals are a key component in MIMO radar, as they allow for simultaneous transmission from all elements without requiring beamforming on transmission [3] These waveforms have very little, or no, cross correlation [13], such that

$$cor[x(t), y(t)] = \int_{-\infty}^{\infty} x(t)y(t)dt = 0 \tag{2.2}$$

where $x(t)$ and $y(t)$ are two time varying signals [3]. Using orthogonal signals allows for all transmitting paths to be independent [3]. Orthogonality in signals can be achieved in numerous ways, one of which is time division multiplexing.

The coherence matrix, $\boldsymbol{R}$, for an M element uniform linear array (ULA) is

$$\boldsymbol{R} = \frac{1}{N}\sum_{n=1}^{N} s[n]s^H[n] = \begin{bmatrix} 1 & \beta_{12} & \cdots & \beta_{1M} \\ \beta_{21} & 1 & \cdots & \beta_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{M1} & \beta_{M2} & \cdots & 1 \end{bmatrix} \tag{2.3}$$

Where $s[n]$ is the transmitted baseband signal vector from M element ULA, n represents the time index, β_{ij} is the complex correlation coefficient between signals i and j and H is the Hermitian operator [14]. The phases of the diagonal elements control the transmitted beam direction; since the signals transmitted are orthogonal the coherence matrix is an identity matrix $\boldsymbol{R} = \boldsymbol{I}$ [14]**.**

Transmitting using orthogonal signals increases the beam width, which results in a greater area of interest (AOI) scanned within each beam compared to phased array radar, but decreases the gain [14].

2.1.1.1 Frequency Modulated Continuous Waveform with Time Division Multiplexing

The orthogonal waveform that is used for transmission in this thesis is a Frequency Modulated Continuous Wave (FMCW) that uses time division multiplexing (TDM). Unlike pulsed radar systems, FMCW radars transmit a continuous wave where the frequency of the transmitted wave is linearly modulated [13]. The FMCW has three distinct advantages compared to a pulse radar system: a constant power envelope, a high Doppler shift tolerance and the ability to use simplified receiver processing [15]. A single chirp FMCW signal (a sinusoidal wave whose frequency increases linearly over time over time) at the transmitter can be represented as [15]

$$s_{Tx}(t) = e^{j2\pi\left(f_s t + \frac{1}{2}\alpha t^2\right)} \tag{2.4}$$

where *t* is the time variable signal within the single chirp period, f_s denotes the start frequency of the chirp, and α is the sweep rate (bandwidth of the chirp divided by the chirp duration). The received reflection from a target within the area of interest is a time-delayed copy of the transmitted signal, where the time delay is related to the range [15]. An example of a transmitted FMCW waveform can be seen in Figure 2.2.

Signal orthogonality is obtained in the time domain by employing TDM, a technique that switches between transmitting elements. The typical TDM switching scheme consists in activating only one transmit antenna at once, starting from the element at position zero and sweeping through the array until all elements have transmitted; upon reaching the end of the array, the scheme is then repeated from the beginning of the array [16]. Using TDM allows for the calculation of amplitude and phase of a large number of points in the area of interest [16].

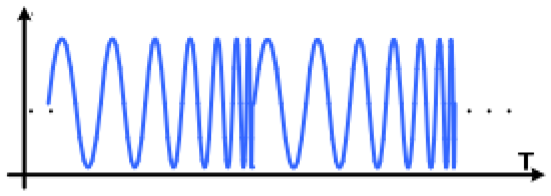


Figure 2.2 FMCW Waveform (reproduced from [17])

2.1.2 Virtual Array

In a MIMO radar, a virtual array is constructed to hold the received data. This concept is an extension of the coarray used for coherent imaging, described in [11]. The virtual array will vary in size depending on the number of transmitting and receiving elements in the radar. If an array of *M* elements are used for transmitting and *N* elements are used for receiving, then the corresponding virtual array has up to *MN* elements; the data located within the array is the return information received by the *Nth* receiver from the *Mth* transmitter [3]. The configuration of the virtual array will be dependent on the spacing between the transmit and receive elements; the elements of the virtual array are formed from the spatial convolution between the transmit and receive elements, represented as [18]:

$$Virtaul\ Array(x) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \delta\big(x - (x_m + x_n)\big) \tag{2.5}$$

where x is the position across the array, x_m are the location of the transmit elements and x_n are the locations of the receive elements. Using a virtual array allows for the system to increase the resolution and accuracy beyond the physical number of elements that are present in the transmit and receive arrays [19]. It also allows for a sparsely filled array to maintain the same resolution of a full array, without the added cost of more hardware [19].

Figure 2.3 illustrates how the virtual array is formed using (2.5). The physical transmit and receive arrays are the top two rows while the bottom row is the corresponding virtual array. In this example, there are three transmitting elements located at (0), (5), and (10) and five receiving elements at (-2), (-1), (0), (1), and (2). Therefore, there are 15 elements in the virtual array after the spatial convolutions of transmit and receive arrays are conducted [3]. The first five virtual array elements are formed through the spatial convolution of the transmitter located at (0) and the five receiving elements; the remaining 10 elements are formed by using the transmitters located at (5) and (10) with the other two receivers. A detailed breakdown of how (2.5) is used to determine locations of virtual array elements can be found in Table 2.1. After the virtual array has been constructed, signal processing is carried out to determine if a target was detected.

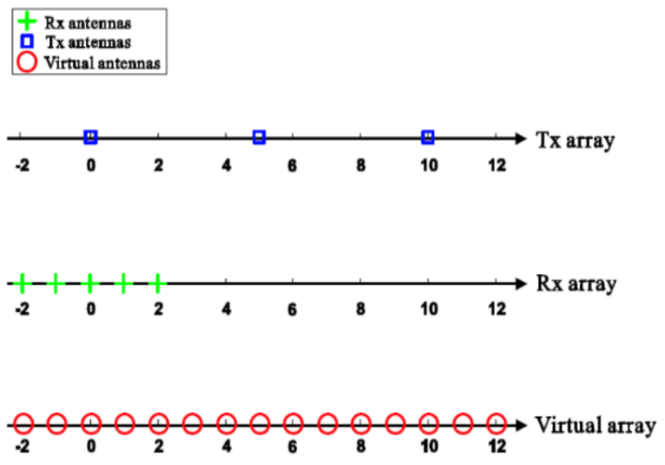


Figure 2.3 An example of a virtual array (reproduced from [3])

Table 2.1 Virtual Array Locations using (2.1)

x_m	x_n	$x_m + x_n$	$\delta\big(x - (x_m + x_n)\big)$	Position
5	-2	3	$\delta(x-3)$	3
5	-1	4	$\delta(x-4)$	4
5	0	5	$\delta(x-5)$	5
5	1	6	$\delta(x-6)$	6
5	2	7	$\delta(x-7)$	7

2.1.3 MVDR Beamforming

An array of receivers, such as the receive array on a MIMO radar, is capable of steering a beam in space by using a process called beamforming [20]. In MIMO radar systems, adaptive beamforming techniques such as the MVDR can be used to obtain the best possible estimation of the desired signal by suppressing the undesired interference and noise components as much as possible [21].

MVDR beamforming is also known as Capon beamforming, which is described in [22]. In MVDR, the signal correlation matrix is used to provide direction of arrival (DOA) estimation [23]. The MVDR technique attempts to minimize the noise and interference by creating nulls toward their directions and maintaining a fixed gain in the look direction [23]. Peaks in the MVDR spectrum occur when the direction vector is orthogonal to the noise [23].

Figure 2.4 denotes a traditional beamformer block diagram. The output signal of the beamformer is given by [24]

$$y(k) = \boldsymbol{w}^H \boldsymbol{x}(k) \tag{2.6}$$

where k is the time index, $\boldsymbol{x}(k) = [\,x_1(k), \dots, x_M(k)]^T$ is the *M x 1* complex vector of array observations, $\boldsymbol{w} = [\,w_1, \dots, w_M]^T$ is the *M x 1* complex vector of beamformer weights, M is the number of array sensors, and $(.)^T$ and $(.)^H$ denote the transpose and Hermitian transpose [24].

The array observation vector, $\boldsymbol{x}(k)$, is given by

$$\boldsymbol{x}(k) = \beta \boldsymbol{s}_s(t) + \boldsymbol{i}(t) + \boldsymbol{n}(t) \tag{2.7}$$

where $\boldsymbol{s}_s(t)$, $\boldsymbol{i}(t)$, and $\boldsymbol{n}(t)$ are independent components of the desired signal, interference, and sensor noise, and β is the primary response parameter, which is typically one.

To find the optimal solution for the beamformer weights,$\boldsymbol{w}$, the signal to interference ratio (SINR) should be optimized. The general form of the SINR is given by [24]

$$SINR = \frac{\boldsymbol{w}^H \boldsymbol{R}_s \boldsymbol{w}}{\boldsymbol{w}^H \boldsymbol{R}_{i+n} \boldsymbol{w}} \tag{2.8}$$

where $\boldsymbol{R}_s$ and $\boldsymbol{R}_{i+n}$ are the M x M signal and interference-plus-noise covariance matrices. Therefore, (2.8) can be optimized by

$$\begin{aligned} &minimize_{\ w}\ \boldsymbol{w}^H \boldsymbol{R}_{i+n} \boldsymbol{w} \\ &subject\ to\ \boldsymbol{w}^H \boldsymbol{R}_s \boldsymbol{w} = 1 \end{aligned} \tag{2.9}$$

The solution to (2.9) can be found by using Lagrange multipliers, and taking the gradient and equating it to zero [24]. Therefore, the optimal weight vector can be written as

$$\boldsymbol{w}_{optimal} = \rho\left\{\boldsymbol{R}_{i+n}{}^{-1} \boldsymbol{R}_s\right\} \tag{2.10}$$

where ρ is the operator which returns the maximal eigenvalue in the matrix [24].

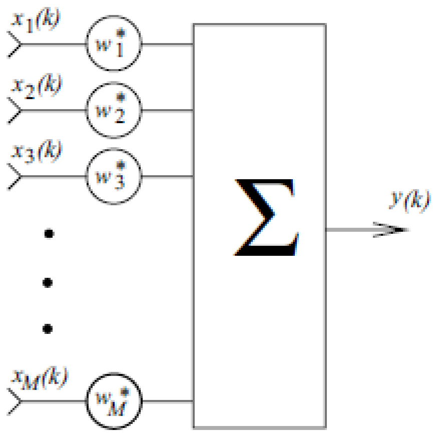


Figure 2.4 Generic beamformer scheme (reproduced from [24])

2.1.4 MVDR Beamforming for MIMO Radar

Since a MIMO radar is a two-channel system and possesses a steering vector which adds constructive interference, the SINR has a slightly different form than (2.8). Therefore, the SINR in a MIMO radar, is defined as [25]

$$SINR = \frac{|\sigma \boldsymbol{w}^H \boldsymbol{s}(\theta_t)|^2}{\boldsymbol{w}^H \boldsymbol{R}_{i+n} \boldsymbol{w}} \tag{2.11}$$

where σ is the radar cross section (RCS) of the target and $s(\theta_t)$ is a steering vector corresponding to the target at location θ_t. To maximize (2.11), Schwartz`s inequality is used and can be re-written as [25]

$$\begin{aligned} SINR &= \frac{\left|\sigma \boldsymbol{w}^H \boldsymbol{R}_{i+n}{}^{\frac{1}{2}} \boldsymbol{R}_{i+n}{}^{-\frac{1}{2}} \boldsymbol{s}(\theta_t)\right|^2}{\boldsymbol{w}^H \boldsymbol{R}_{i+n} \boldsymbol{w}} \\ &\leq \sigma^2 \frac{\boldsymbol{w}^H \boldsymbol{R}_{i+n} \boldsymbol{w} s^H(\theta_t) \boldsymbol{R}_{i+n}{}^{-1} \boldsymbol{s}(\theta_t)}{\boldsymbol{w}^H \boldsymbol{R}_{i+n} \boldsymbol{w}} \\ &\leq \sigma^2 \boldsymbol{s}^H(\theta_t) \boldsymbol{R}_i + n^{-1} \boldsymbol{s}(\theta_t) \end{aligned} \tag{2.12}$$

Assuming that σ= 1, the optimal $\boldsymbol{w}$ that produces the maximum value of (2.12) is [25]

$$\boldsymbol{w} = \frac{\boldsymbol{R}_{i+n}{}^{-1} s(\theta_t)}{\boldsymbol{s}^H(\theta_t)\boldsymbol{R}_{i+n}{}^{-1}\boldsymbol{s}(\theta_t)} \tag{2.13}$$

In practice, it is difficult to obtain $\boldsymbol{R}_i$, instead, the sample covariance matrix of the receiver, $\boldsymbol{R}_{xx} = \sum_{n=1}^{N} x_n x_n^H$, is used where x_n are data snapshots which are collected from *N* different radar pulses. Therefore, (2.13) becomes [26]

$$\boldsymbol{w} = \frac{\boldsymbol{R}_{xx}{}^{-1} s(\theta_t)}{\boldsymbol{s}^H(\theta_t)\boldsymbol{R}_{xx}{}^{-1}\boldsymbol{s}(\theta_t)} \tag{2.14}$$

Capon has shown in [22] that the MVDR spectrum can be found by

$$S = \frac{1}{\boldsymbol{s}^H(\theta_t)\boldsymbol{R}_{xx}{}^{-1}\boldsymbol{s}(\theta_t)} \tag{2.15}$$

where *S* is the MVDR spectrum. Since it is not possible to carry out division on a matrix, (2.17) is computed by finding an inverse. Therefore, it can be re-written as

$$S = \left(\boldsymbol{s}^H(\theta_t)\boldsymbol{R}_{xx}{}^{-1}\boldsymbol{s}(\theta_t)\right)^{-1} \tag{2.16}$$

This thesis is interested in exploring the MVDR spectrum for its applicability to target detection. Therefore, (2.16) will be used as a basis to create an algorithm to extract targets from the MVDR spectrum.

2.2 GPUs

Starting around 1980, CPU innovation produced rapid performance increases for almost 20 years. These performance increases have allowed greater software functionality, and improved user interfaces [13]. However, this in turn has also made users place greater demands for further development once they have become accustomed to the improvement.

Human-generated code has traditionally been written as a serial program. The sequential code can be understood easily by humans, as we can step through each line of code at a time. Because of the rapid hardware developments made until the early 2000s, developers have been relying on hardware advances to improve software speed; with a faster microprocessor the same code will simply run faster [2]. Sequential (or serial) code will only run on one processor core instead of several. Since we have reached the limit of processor innovation, new ways must be developed to achieve performance increases. An example is the development of the GPU. Instead of having a small number of processor cores available to compute serial code, a GPU has hundreds (possibly even thousands) of cores available to compute parallelized code. Parallelizing code can take some effort as it is not as intuitive as serial code, however the extra development time is acceptable when a speed up is possible.

Originally developed for the video game industry, GPUs are now being used for a multitude of purposes, such as parallel execution of computationally intensive algorithms (i.e. matrix inversion). General Purpose Graphic Processor Units (GPGPUs) are now being used in applications which require intense computing ability due to its unique architecture.

2.2.1 GPU Architecture/Programming Model

A GPU can carry out parallelized processing based on its architecture. A GPU has multiple stream multiprocessors (SMs), which have a certain number of cores [2]; the exact amount of SMs and cores on a GPU depends on the exact model of GPU used.

In a GPU, there are five specific types of memory available to the programmer: registers, global memory shared memory, constant memory, and texture memory [2]. Registers are dynamically allocated and are private to each thread and provide the fastest access to data [2]. Global memory, also known as off-chip memory, is accessible to all threads/thread blocks with a large capacity, however the read/write access time can be slow [2]. Shared memory, also called on-chip memory, is accessible by all threads within a block. Access time for shared memory is approximately 100 times faster than that of global memory, however it has significantly less capacity than that of global memory [2]. Constant memory and textured memory are both located off-chip, however they are both cached [2]. Therefore, these types of memory are also significantly faster than global memory. Variables and arrays used within kernel launches can be stored in any of these five memory types [2]; the type of memory chosen will depend on the algorithm that is to be executed.

GPUs contain cores which execute threads. These are located within a thread block, which are located within a grid [10]; blocks and grids can either be one, two, or three dimensional [2]. Figure 2.5 displays a GPU architecture with N threads, a three-dimensional block size of (x, y, z) and a one-dimensional grid size.

Threads are the basic unit in parallel programming. Threads within a block can synchronize their behavior and communicate with each other [2]. Threads are managed, scheduled, and created in groups of 32 which are known as warps [2]. The threads that will execute the parallel code are created when a kernel launch to that parallel code is invoked [2]. Each thread can select data using its' unique thread identification (ID) and is then able to execute the programmed algorithm with its selected data and store it in the appropriate location [10].

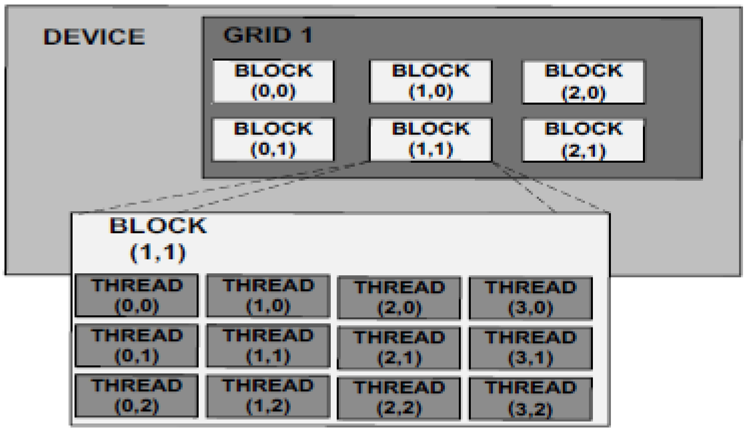


Figure 2.5 GPU thread and block model with N threads and (x ,y, z) block size (reproduced from [10])

2.2.2 Heterogeneous Computing

Due to the fundamental hardware difference between a CPU and a GPU, there are some scenarios where a CPU would be better suited than a GPU based on design and architecture. The design of a CPU is optimized for peak sequential code performance and makes use of sophisticated logic to allow instructions to be carried out in parallel, but still maintaining the sequential flow of the program [2]. The most important design to highlight is the CPU’s large memory cache, which reduces latency in instructions and data memory access [2]. GPUs are designed to optimize execution throughput by efficiently utilizing their many cores to execute parallel operations [27]. The multi-thread GPU floating-point calculation throughput is approximately 10 times that of a multicore CPU [2].

CPUs are designed to minimize the execution time of a single thread. Its large on-chip caches are designed to capture frequently accessed data into short-latency cache access [2]. The arithmetic logic units (ALUs) are also designed to minimize operation latency, which increases their use of the available chip area [2]. By reducing the latency of single thread operations, the CPU hardware reduces the overall latency of a single thread [2]. However, the CPU large cache memory, low-latency ALUs and sophisticated logic consume much of the available chip area and power [2].

GPUs were designed to optimize the execution throughput of a massive number of threads. This design reduces the overall chip area and power by allowing memory channels and arithmetic operations to have long latency [2]. The reduced area and power allows designers to have more hardware on a chip, increasing total execution throughput of the GPU [2]. Therefore, GPU software applications are expected to be written to maximize the number of threads that are used within the operation.

From this, it is possible to deduce that CPUs will provide better performance that have low numbers of threads or a small data set; GPUs will provide a higher execution throughput when a program has a very large number of threads or a massively large data set [2]. Therefore, there will be many applications where both a CPU and a GPU should be used in tandem to achieve the best possible performance.

Figure 2.6 highlight the difference in GPU and CPU architecture. From this Fig, it is possible to visually see the difference in architecture described above. The CPU has a greater cache, and four ALUs and the controller take up approximately 80% of available chip space [2]. Conversely, the GPU has a small cache available for multiple ALUs, but there are a significantly greater number of ALUs. Each ALU on the GPU is also significantly smaller small those on the CPU; however, all the ALUs take up approximately 90% of the chip space [2]. By looking at Fig 4, it can be rationalized that a CPU would perform better with very few threads while a GPU would have better performance with a massive number of threads.

Heterogeneous computing exploits the advantages of both CPUs and GPUs to achieve the fastest possible program. In this form of programming, code is written in a pre-existing serial programming language (i.e. C, Python, and FORTRAN). The CPU will run the serial code, until a call to the GPU code is detected. Once the GPU work is completed, the CPU will continue the serial execution until the program is completed or the GPU is required again [2] . GPUs are typically assigned computationally intensive algorithms to reduce the workload on the CPU, while basic instructions and code is executed on the CPU due its low latency. The combination of sequential and parallel programming in one overall program is done using a programming extension called Common Unified Developer Architecture (CUDA).

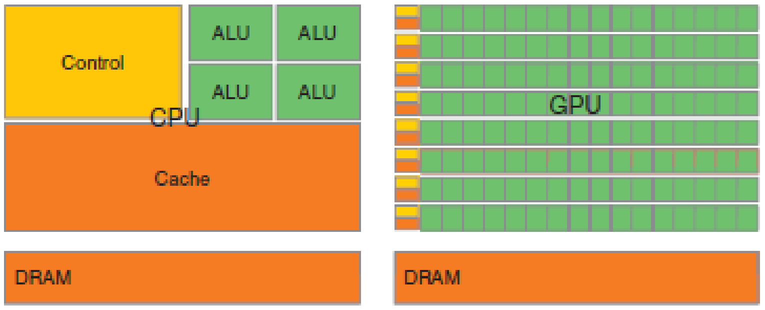


Figure 2.6 GPU vs CPU architecture (reproduced from [2])

2.2.3 Common Unified Developer Architecture

CUDA is a programming language that exploits the parallel processing capabilities of GPUs and is used in heterogeneous computing; CUDA can be used in conjunction with the C programming language in order to write parallel programs [6]; NVIDIA has developed CUDA extensions that can be used with other languages, such as FORTRAN and Python [28]. A CUDA C program contains a mixture of sequential (CPU) and parallel (GPU) code, where the CPU is known as the host and the GPU is known as the device [2]. In the program, the programmer writes a kernel which at launch time will execute the portion of the code which will be carried out on the GPU [10]. After the kernel is complete, it then returns to the host so that the host can continue executing the program.

Since CUDA is compatible with several different programming languages, the coder can use a language which he or she is familiar with; this decreases the learning curve associated with parallel programming.

2.3 GPU Programming in MATLAB®

The simulation created and described in Chapters 3 and 4 use standard MATLAB® commands, as well as the Parallel Computing Toolbox™ and Phased Array Toolbox™. These toolboxes allow for the use of a GPU to execute parallel code within a MATLAB® script and provides algorithms for the design and simulation of sensor array systems.

2.3.1 Parallel Computing Toolbox™

The Parallel Computing Toolbox™ allows the user to parallelize MATLAB® applications and code without having to use CUDA programming [29]. Using this Toolbox™, the user can create traditional MATLAB® code and parallel code within the same simulation.

A major advantage to this toolbox is not having to use CUDA in conjunction with a programming language to execute parallel code. Programming in CUDA requires the user to code in a low level programming language (i.e. C, python) and to create numerous portions of code, including but not limited to specifying block size, specifying grid size, allocating/deallocating memory, determine in which type of memory the algorithm will execute, copy data back and forth from the CPU and GPU, and creating a kernel which will switch execution from CPU to GPU. Since MATLAB® is a high-level programming language, all the overhead required to set up a parallel program is done behind the scenes without user programming. Users who are already familiar with MATLAB® and its programming language will have an advantage, as many of the programming commands can be used to manipulate data once it has been copied to the GPU.

This strength is also one of the toolbox's limitations. Using MATLAB®, programmers are unable to use low-level GPU features, such as shared memory. The programmer also cannot attempt to manipulate block size or grid size, which could also increase the overall speedup obtained. The inability to dictate in which memory space the calculations will occur also limits the speedup obtained. Since shared memory is approximately 100 times faster than global memory, it is advised to utilize shared memory as much as possible which is not possible using MATLAB®.

It is possible to combine C and CUDA to utilize shared memory through MEX files [30]. Through MEX files, programmers can create their own independent C, C++, or FORTRAN subroutines and then execute them in MATLAB® as if they are built in MATLAB® functions [31]. Using this capability requires creating the function in a C, C++, or FORTRAN source file, installing a compatible compiler, creating a gateway function in one of the source files, and using the MATLAB® `mex` command to create a build script which will create a binary MEX file [32]. This MATLAB® capability was not explored in this thesis.

Even with MATLAB®'s limitations, it was decided to use it to program and run the simulations coded within this section. This decision was made primarily because of MATLAB®'s simplified programming structure. Using shared memory could potentially increase the speedup obtained, however obtaining the fastest possible execution speedup was not the main goal of this thesis; successful implementation of the designed MVDR algorithm along with an overall system speedup was the main achievements that were sought.

2.3.2 Phased Array Toolbox™

The Phased array Toolbox™ provides code for the design, simulation, and analysis of array systems in radar applications [33]. This toolbox allows the user to design and create phased arrays and analyze their performance using data of their choice [33].

Using this toolbox provides the user with a straightforward, practical ability to create and manipulate phased arrays and aspects associated with phased arrays. In the simulations described in this chapter, this toolbox is used mainly to create the ULA, the transmit arrays, the received arrays, and the steering vector.

3 Multi-Target Detection using ULA

To ensure that the developed MVDR algorithm functioned correctly, the first simulations that were performed used ULAs. A ULA is an array of sensor elements which are arranged along a line in space with uniform spacing between elements [34]; a MIMO radar uses ULAs for its transmit and receive arrays. ULAs allow for a simpler transmit/receive scheme than a MIMO radar simulation, allowing for easier debugging of the algorithm.

3.1 System Model

The first simulation that was designed used a ULA to detect signals using the MVDR spectrum; the MATLAB® code can be found in Appendix A Section A.1. A ULA was chosen as the starting point as the system is much less complex than a full MIMO radar setup; starting with a simpler system allows for greater ease during the debugging process. It was solely created to ensure accurate target detection of the designed MVDR algorithm; decreasing the algorithm's execution time was not the primary goal of this simulation creation.

In this scenario, targets are pre-positioned at an angle relative to zero degrees from a ULA. Returns are collected from these assumed DOAs and processed using the MVDR algorithm for target detection; a pictograph of this setup can be found in Figure 3.1. This figure is for visual purposes only and is not to scale for the simulation that is described in this section.

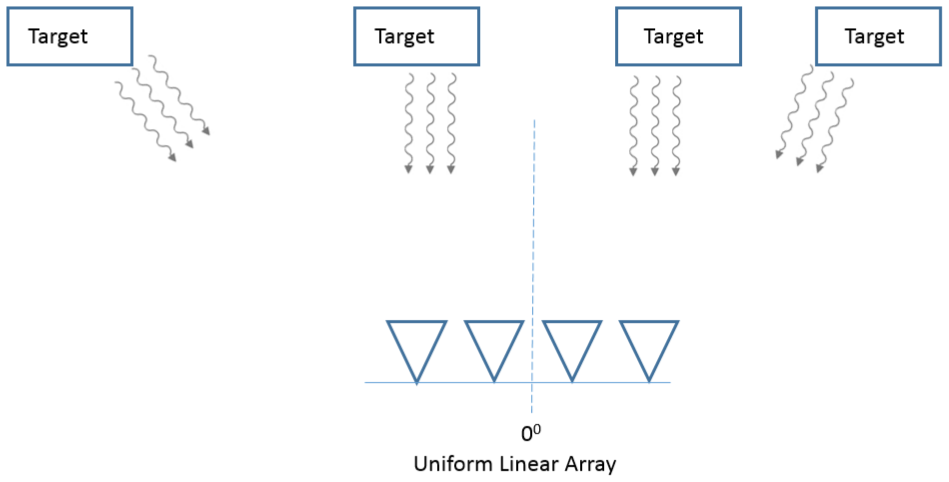


Figure 3.1 ULA Simulation Diagram

The first step taken to simulate this system is to define the required operating parameters, create the required ULA, and simulate the systems with static targets. The chosen operating parameters can be seen in Table 3.1. A basic ULA with 0.5λ element spacing was created. The frequency associated with each of the sine wave inputs outlined in Table 3.1 is the carrier wave frequency of each signal.

Four input sine waves were chosen, all occurring at different frequencies and direction of arrival angles. The DOA is measured from the center of the ULA, where the center is considered an angle of zero degrees. A negative degree value would be on the left side of the array, where a positive degree value would occur on the right side of the array. The MATLAB® function `collectPlaneWave()` was used to simulate the system. This function returns the received signals of the ULA from the input signals that arrive from a given direction; in this function it is assumed that the input signals are plane waves [35].

Table 3.1 ULA Characteristics

ULA Operating Frequency	300 MHz
Wavelength	0.99m
Input Signals	100 Hz sine wave
	200 Hz sine wave
	300 Hz sine wave
	400 Hz sine wave
Signal DOA	-40 degrees
	-20 degrees
	10 degrees
	30 degrees

3.2 MVDR Algorithm

Although MATLAB® has a built in MVDR function (`mvdrweights()`), a custom algorithm was developed for use in this simulation. The MATLAB® function returns the beamformer weights for a phased array; these weights must then be applied to the array to steer the array response in a specific arrival direction [36]. Since this thesis is interested in examining the MVDR spectrum for target detection, a custom MVDR algorithm needed to be developed.

The MVDR algorithm outlined in Section 2.1.3 and 2.1.4 describes the MVDR algorithm in general and with respect to MIMO radar. Since this thesis will focus on the MVDR spectrum, (2.16) will be used as the backbone of the developed MVDR algorithm.

The steering vector for the ULA is created using a MATLAB® built in function, `phased.SteeringVector()` and the covariance matrix of the received signals is determined. Initially, the `cov`() function in MATLAB® was used to determine this matrix. However, it was found that using this function to produce the covariance matrix gave inconsistent data between the CPU and GPU. From the `cov()` documentation provided by MATLAB®, the covariance matrix is calculated using the same equation and algorithm regardless of data type or operating platform [37]. Due to this, it is assumed that there is a problem within the `cov()` source code that causes incorrect calculations or memory overwrites while executing on the GPU. It is unclear exactly what caused the different covariance matrices, as the `cov()` command is a well-tested MATLAB® function; without the ability to drill far enough into source code to debug this function, it is difficult to determine the exact cause of the error.

Since the `cov()` command could not provide fidelity in its calculations, the covariance matrix was instead calculated by

$$\boldsymbol{R}_{xx} = (\boldsymbol{x}') * \boldsymbol{x} \tag{3.1}$$

where $\boldsymbol{x}$ is a matrix containing the received signal returns, and $\boldsymbol{x}'$ is the complex conjugate transpose of the matrix containing the received signal returns. Once the steering vector and the covariance matrix have been determined, the algorithm commences.

The MVDR algorithm that was designed manipulates the steering vector and the covariance matrix to evaluate the spectrum for target direction determination. In MATLAB®, the following code was written to translate (2.16) into code:

```
S = inv(transpose(sv)*inv(Rxx)*sv)'            (3.2)
```

where $\boldsymbol{S}$ is the MVDR spectrum, `sv` is the steering vector, `transpose(sv)` is the non-conjugate transpose of the steering vector, `inv(Rxx)` is the inverse of the covariance matrix, and `'` is used to determine the complex conjugate transpose. The complex conjugate transpose is required to negate the imaginary parts of the complex numbers within $\boldsymbol{S}$.

Using (3.2) provided accurate target detection on the CPU, however it was found that on the GPU, erroneous targets were found. Upon closer review, it was found that the spectrum determined using (3.2) did not produce the same matrix inverse using the `inv()` MATLAB® command. The MATLAB® documentation provided for the `inv()` command states that `inv()` performs a lower-upper (LU) decomposition of the matrix and then uses this decomposition to form a linear system whose solution is the matrix inverse [38]. A LU decomposition is a factorization; it is a way of decomposing a matrix into an upper rectangular matrix, $\boldsymbol{U}$, a lower rectangular matrix, $\boldsymbol{L}$, and a permutation matrix, $\boldsymbol{P}$ [39]. These matrices describe the steps needed to perform Gaussian elimination on the matrix to solve the system of equations. For the GPU to calculate a different inverse than the CPU, the solution to the system of equations created and solved by the `inv()` command must be different. Unfortunately, it is unknown where the discrepancy between the CPU and GPU solution exists, and the user is unable to delve deep enough into MATLAB® source code to pinpoint the exact cause of the error.

With a different solution found on the GPU, a different mathematical solution for the MVDR spectrum was required. According to (2.7), the covariance matrix can be written as

$$\boldsymbol{R}_{xx} = \boldsymbol{S}_v \boldsymbol{R}_{ss} \boldsymbol{S}_v^H + \boldsymbol{R}_{i+n} \tag{3.3}$$

where $\boldsymbol{R}_{ss}$ is the signal diagonal covariance matrix. $\boldsymbol{R}_{xx}$ can be diagonalized as follows:

$$\boldsymbol{R}_{xx} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^H \tag{3.4}$$

where $\boldsymbol{V}$ is a unitary matrix containing the eigenvectors, and $\boldsymbol{\Lambda}$ is a diagonal matrix containing the corresponding eigenvalues. For high SINR we can assume the following approximations: $\boldsymbol{\Lambda} \approx \boldsymbol{R}_{ss}$ and $\boldsymbol{V} \approx \boldsymbol{S}_v$. It follows that the MVDR spectrum can be simplified as [40]

$$\boldsymbol{S} = \boldsymbol{diag^{-1}}\{\boldsymbol{S}_v^T R_{xx}^{-1} \boldsymbol{S}_v\} \tag{3.5}$$

and the following code was then used to determine the spectrum in decibels:

```
S_dB = -10long(transpose(sc)*(Rxx\sv))        (3.6)
```

It should be noted that (3.6) is valid only for high SINR. On the other hand, if there is a strong interference, additional processing, such as the diagonal loading compensation [41], is required. For this thesis, we assume the receiver treats interference as noise. The `\` operator used in (3.3) executes the `mldivide()` command in MATLAB®. This operation solves the following linear system of equations

$$\boldsymbol{A} * \boldsymbol{x} = \boldsymbol{B} \tag{3.7}$$

so that

$$\boldsymbol{x} = \boldsymbol{A}/\boldsymbol{B} \tag{3.8}$$

where $\boldsymbol{A}$ and $\boldsymbol{B}$ are matrices with the same number of rows [42]. The exact algorithm used to solve this system of equations varies depending on numerous factors: whether the arrays are dense or sparse, dimensions of the matrices involved, and the numerical values found along the matrix diagonal [42]. The exact algorithm that MATLAB® chooses for the execution is hidden from the user as this logic flow is found within MATLAB®'s source code.

There is an inherent speedup in using (3.6) versus (3.2), as a full matrix inversion does not need to be calculated. Matrix inversions are computationally intensive and will slow down a program; by removing the inversion a speedup on both the CPU and GPU would be seen when comparing (3.6) to (3.2).

(3.6) was then tested using data produced by both the CPU and the GPU. After careful comparison between CPU and GPU calculations of the spectrum, the results were found to be identical. When the spectrum was plotted, identical results were found, indicating that (3.6) was providing the required functionality and operating correctly

3.3 Results and Analysis

3.3.1 Hardware and Software Specifications

The simulations run and the results obtained within this section were all completed using the same computer. This was done to ensure that any differences in hardware across platforms would not have an impact on execution times. Using a different computer with different hardware could produce a different execution time for each scenario.

For these simulations, a Microsoft computer running Windows 7 with 32 GB of random-access memory (RAM) was used to run MATLAB® 2018b. The GPU used to carry out the parallel algorithm is a NVIDIA GeForce GTX TITAN. Specifications for this GPU can be found in Table 3.2 [43], [44], [45]. This GPU is based on the Kepler Architecture [28].

Computing capability is represented by a version number [28]. This number (in the case of the GPU used in this thesis, 3.5) identifies features that are supported by the GPU hardware and used at runtime to determine which features are available to the GPU [28]. GPUs which have the same computing capability number have the same architecture [28].

Table 3.2 NVIDIA GeForce GTX TITAN Operating Specifications

CUDA Cores	2688
Base Clock	837 MHz
Boost Clock	876 MHz
Memory Clock	6.0 Gbps
Memory Bandwidth	288.4 GB/sec
Computing Capability	3.5
Max Threads/Warp	32
Max Threads/Multiprocessor	2048
Max Blocks/Multiprocessor	16
Max Shared Memory/thread block	48 kilobytes
Max Registers/block	65536
Max X Grid Dimension	2^{32}-1

3.3.2 Simulation Results

The simulation design used to obtain the results discussed in this section is described in Section 3.1. Four targets, placed at -40, -20, 10, and 30 degrees were used in order to determine accurate target detection.

The first simulation used eight elements in the ULA array. In this simulation, it was determined that the CPU developed algorithm was successful in detecting all targets; the MVDR spectrum showing the correct target detection is found in Figure 3.2. The GPU was also able to successfully detect all targets; its MVDR spectrum can be found in Figure 3.3. In both figures, peaks can be seen at -40, -20, 10 and 30 degrees. These peaks represent each of the targets that were defined at the beginning of the simulations. Since these peaks occur at the angles where the targets were defined, it is inferred that the both the serial and parallel algorithms were programmed correctly and function as desired.

The mean square error (MSE) was also evaluated between the two MVDR spectrums calculated on the CPU and the GPU. The overall MSE found for an eight element ULA system was 1.084 x10^{-7} and the root MSE was 3.30 x10^{-4}, which is the average error between both sets of data. Figure 3.4 shows the MSE over all angles within the spectrum. The MSE fluctuates most around the angles at which the four targets occur; this is expected since this is where the calculated differences would be expected to be the greatest. These small errors highlight that the MVDR algorithm is functioning correctly on the GPU, as we would expect given the visual indication found in Figure 3.2 and Figure 3.3.

With the developed algorithm functionality verified, the number of elements in the ULA was gradually increased. Table 3.3 details the CPU and GPU execution times of the simulations run with varying numbers of ULA elements, as well as the execution speedup that was obtained; Figure 3.5 shows a pictographic view of Table 3.3's result.

The results that are present from both Table 3.3 and Figure 3.5 are what one would expect to see from this basic simulation setup. The simulation results show that the GPU performed the developed MVDR algorithm faster than the CPU version in all simulations that were run. Since the GPU can execute multiple tasks at once, it is expected that the GPU would execute the algorithm faster than the GPU. An anomaly is observed when eight elements are used in the ULA. When eight ULA elements are simulated, the MVDR algorithm takes approximate 2.6 times longer than when 16 elements are simulated. This is unexpected, as it is assumed that a simulation with less elements should be faster. This anomaly can be explained since the eight element ULA does not have enough data to take advantage of the parallel capabilities of the GPU. As outlined in Section 2.2.2, GPUs work by maximizing the efficiency of a massive number of threads; a single thread will have higher latency on the GPU then the CPU as a result. When the eight element ULA is executed on the GPU, the number of threads launched and used to carry out the algorithm are not enough to reduce the latency of each thread so that the execution time is reduced.

With the additional elements in the 16-element array and the increase in data, the algorithm processing is more efficiently distributed across the GPU cores which allows for a decreased execution time. The exact number of threads launched in the eight and 16 element ULA is unable to be determined due to MATLAB® limitations.

It should be noted that the relative efficiency gained by using a parallel implementation of the MVDR algorithm with a ULA is small regardless of the chosen element size. The mean speedup found during this analysis was only 1.8 times. In a real-world scenario where a ULA would be used with this algorithm, it would not be recommended to create and execute a GPU version of the algorithm. While there is a speedup obtained (1.11 x, 2.7x, 1.78x, 1.87x, and 1.78x), the execution time decrease is minimal, even when 128 array elements are used. The small decrease in execution time is negated by the increased effort by the programmer to create parallel code.

Even though the code is only written once and can be used by a fielded system many times, it takes much more effort on the part of the programmer to design, create, and debug a parallel program. The creation of parallel code is not an intuitive process and requires a different program flow and programming than serial code. It is not possible to do a tradition debug process where the programmer can step through each line of code to verify execution. On the GPU, it is not possible to guarantee the order of executions. Thread blocks do not execute in sequential order; their order of execution is random and unpredictable, and it is not possible to view data in memory once it is on the GPU. Therefore, the only option for debugging purposes is to create data transfers between the CPU and GPU to verify data. This creates a higher overhead cost during the development process and more time requirements from the programmer. It must be evaluated that the higher overhead and longer programming time is worth the potential speedup obtained. In the case of the ULA system that is simulated in this chapter, it would not be recommended as the speed shown in the system is too small (mean speedup of only 1.7x) to warrant the extra programming overhead.

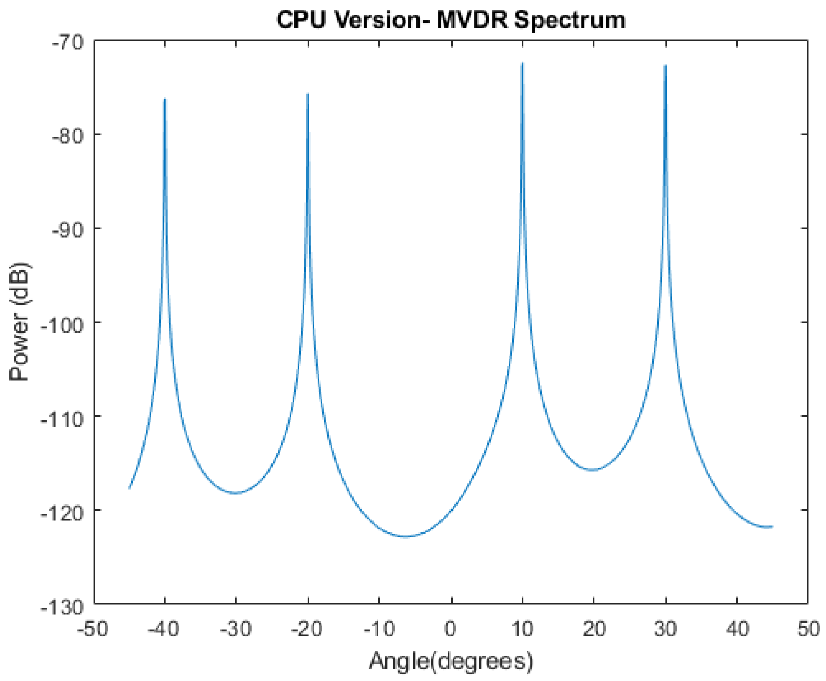


Figure 3.2 CPU MVDR Spectrum with eight ULA elements

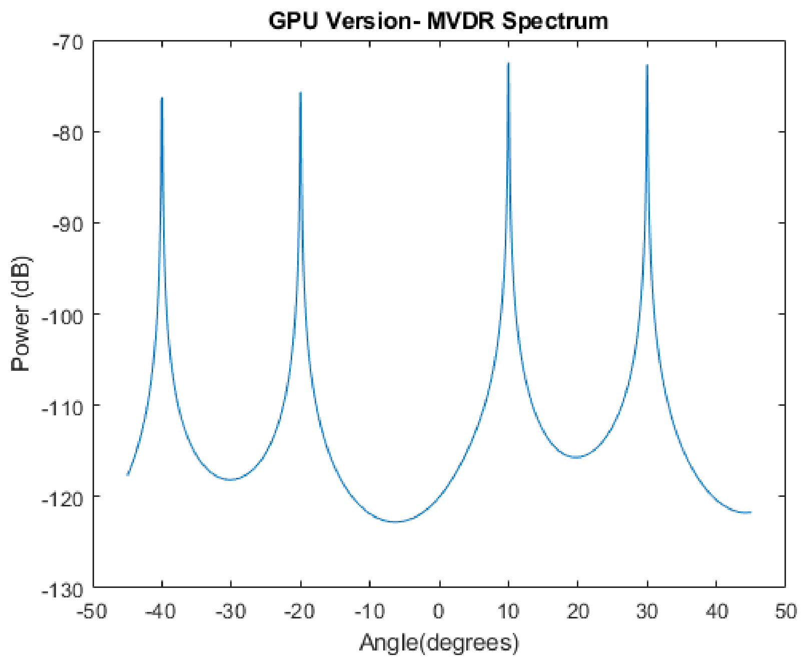


Figure 3.3 GPU MVDR Spectrum with eight ULA elements

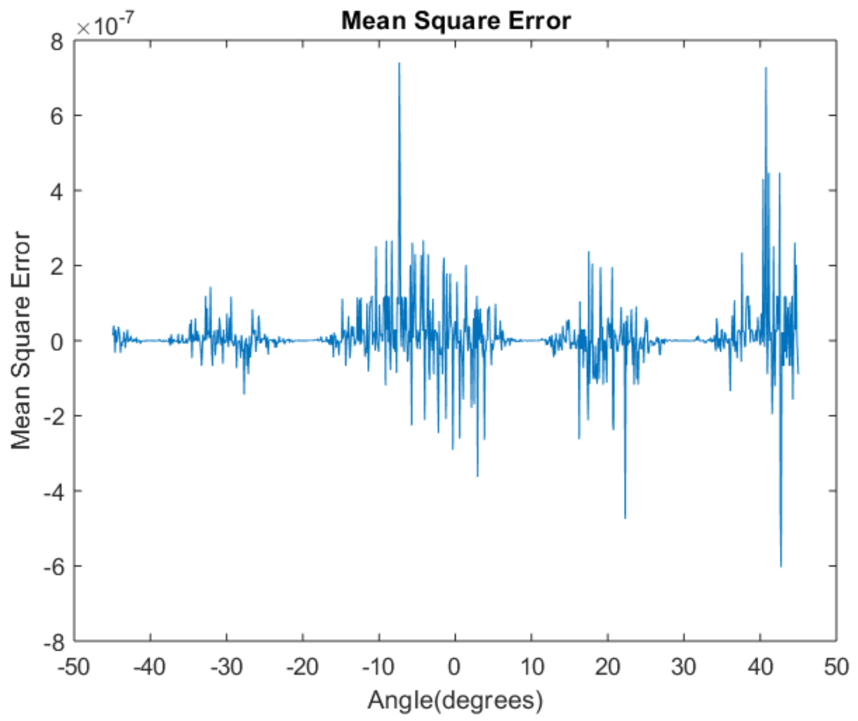


Figure 3.4 MSE between CPU and GPU data with a ULA of eight elements

Table 3.3 CPU and GPU execution times and execution speedup for various ULA elements

Number of Elements	**CPU Execution Time (ms)**	**GPU Execution Time (ms)**	**Execution Speedup**
8	25.772	23.307	1.11x
16	23.837	8.824	2.70x
32	24.356	13.724	1.78x
64	29.988	16.036	1.87x
128	44.51	24.889	1.79x

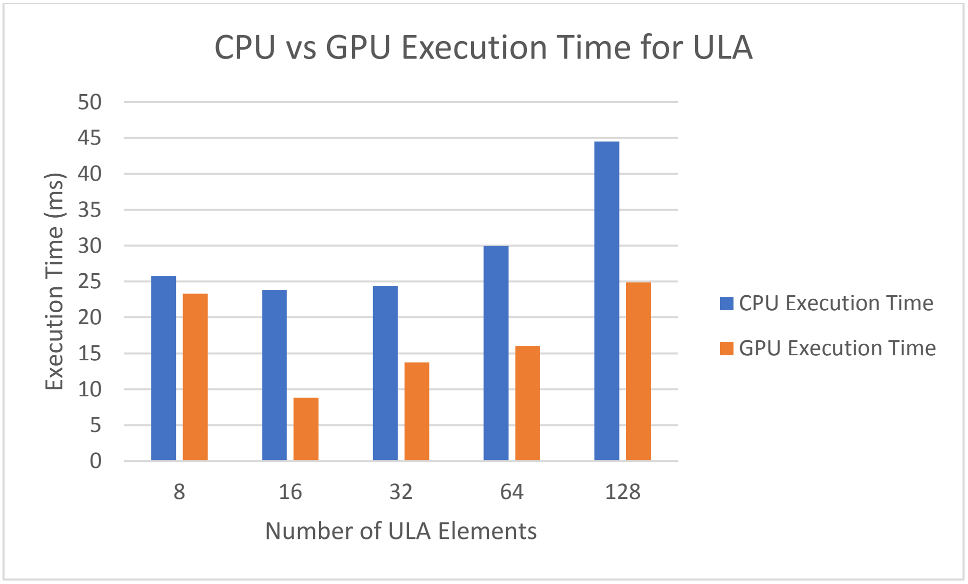


Figure 3.5 CPU vs GPU Execution Times for various ULA elements

4 MIMO Radar Simulation

Since the target acquisition MVDR algorithm was found to function accurately, a true MIMO radar setup using two moving cars as targets was then simulated. The MATLAB® code created to run this simulation can be found in Appendix A Section A.2. The simulation created in this chapter is based off the MATLAB® example "Increasing Angular Resolution with MIMO Radars" [46]. The radar is a MIMO radar which is onboard an autonomous vehicle; there are two moving vehicle targets. The parameters used in this simulation are based off those used for automotive adaptive cruise control (ACC).

4.1 System Model

In this simulation, the transmit array on the car transmits a signal. This signal encounters the targets in the AOI, which then returns an echo to the receive array. This data is then transferred to the virtual array and processed for target detection. A visual representation of how this process works using vehicles can be seen in Figure 4.1; a visual representation of this focusing on the radar operation can be seen in Figure 4.2. In Figure 4.2, the element- by-element transmission scheme that is described in Section 4.1.4.2 is shown by each transmitting element having its own beam; in the simulation however, only one transmitting element will transmit at a time. These figures are for visual purposes only and are not to scale for the simulation that is described in this section.

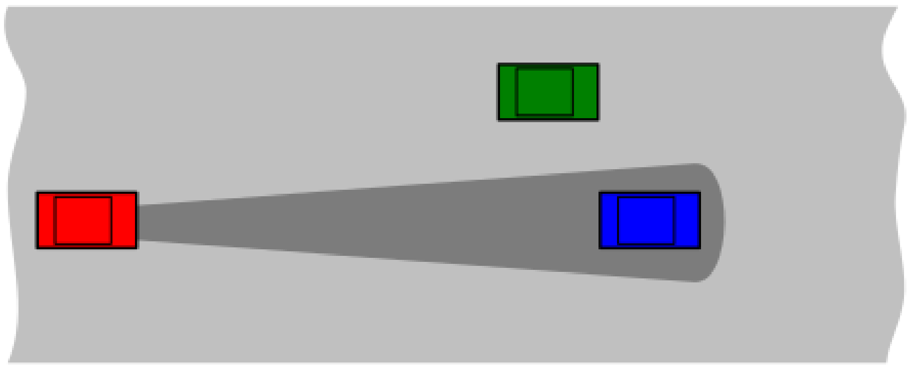


Figure 4.1 ACC (reproduced from [47])

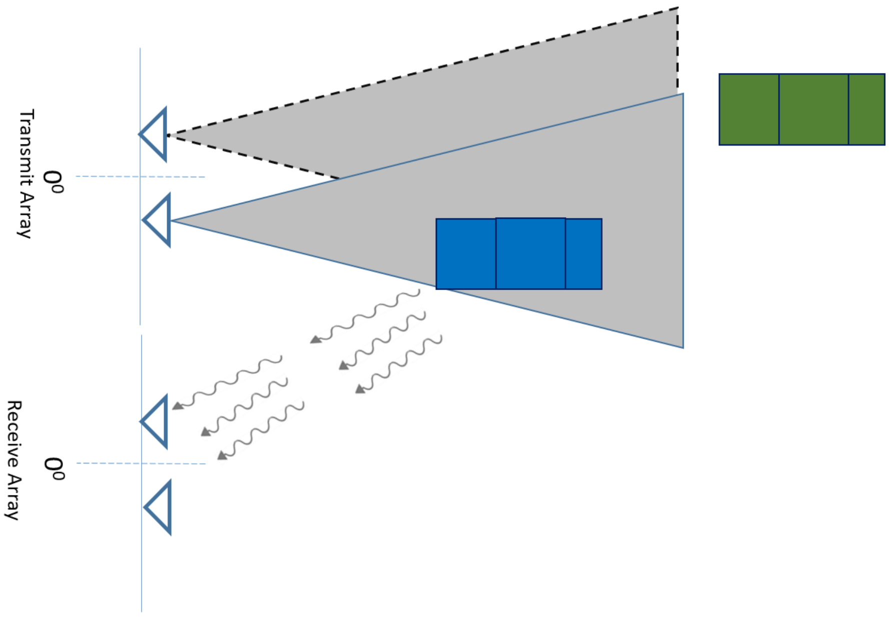


Figure 4.2 MIMO Radar ACC (Radar View)

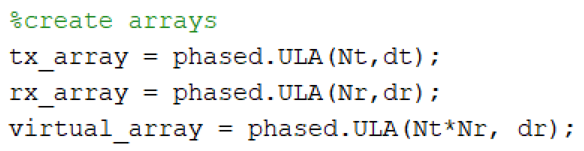
In order to accurately simulate the MIMO radar system, there are various system parameters which must be set. The parameters chosen for this simulation can be found in Table 4.1. This is MIMO radar setup differs from a multistatic radar, as all of the data received by the receive array is jointly processed to determine if a target has been detected; this is done by using the virtual array (discussed in Section 4.1.4.3) [48]. In a multistatic radar, signal processing is carried out by each independent receive radar in the multistatic system, with a central unit that will then process the output of each receiver system to determine if a target is present [48].

Table 4.1 MIMO Radar System Operating Parameters

Sampling Frequency	8000 Hz
Operating Frequency	300 MHz
Wavelength (λ)	0.99 m
Transmit Array Spacing	0.5λ
Receive Array spacing	0.5λ

4.1.1 Transmit/Receive/Virtual Array Creation

Once the defining characteristics of the simulation are set, the transmit, receive, and virtual arrays are created. All arrays are created using the `phased.ULA()` MATLAB® function; the code used to create these arrays can be found in Figure 4.3. This function is embedded within the Phased Array Toolbox™ , and will create a ULA with the desired number of elements and the required array element spacing [34]. To create the transmit and receive arrays, the number of transmit and receive elements that is desired is used in `phased.ULA()` for the desired number of elements. However, the virtual array is created slightly differently. Since the virtual array is created by convolving the transmit and array antennas together, the elements of the transmit and receive arrays are multiplied together. The array spacing used for the virtual array is that of the receive array.



```
%create arrays
tx_array = phased.ULA(Nt,dt);
rx_array = phased.ULA(Nr,dr);
virtual_array = phased.ULA(Nt*Nr, dr);
```

Figure 4.3 Array Creation Code

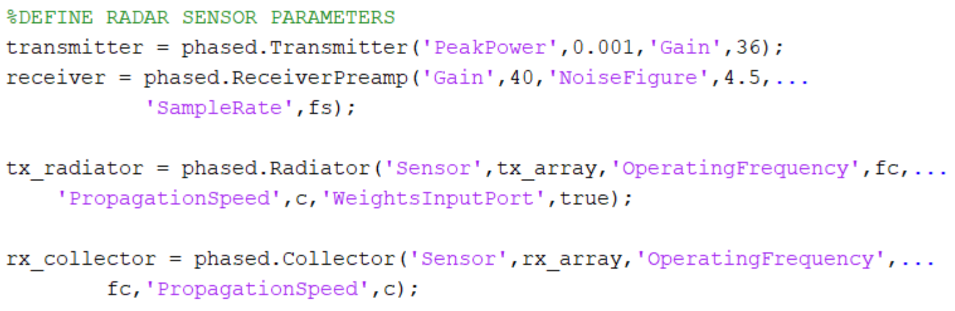
4.1.2 Radar Sensor Parameters and Setup

Since the simulation in this scenario is not using the `collectPlaneWave()` MATLAB® function to simulate targets, it is necessary to define the operating characteristics of the transmitting and receiver elements. These parameters are outlined in Table 4.2.

Table 4.2 Radar Sensor Parameters

Transmitter Peak Power	0.001 W
Transmitter Gain	36 dB
Receiver Gain	40 dB
Receiver Sample Rate	8000 Hz
Receiver Noise Figure	4.5 dB

With the transmitter and receiver operating characteristics defined, the transmit and receive array then need to be set as a radiator and a collector. In a physical MIMO radar setup, it is possible to transmit and receive simultaneously; an array can be both a radiator and a collector. However, since this design is completely software based, it is required to define a separate array as a radiator (the transmit array) and another array as a collector (the receive array). The designation of the radiator and collector, along with the required parameters, is done using the `phased.Radiator()` and `phased.Collector()` commands available from the Phased Array Toolbox^{TM}. The code that carries out this step can be seen in Figure 4.4.The virtual array which is created and defined in Section 4.1.1 does not apply to this section/ Figure 4.4 as the virtual array is not used for receiving the radar returns from the AOI; the data is transferred to the virtual array once it has been received by the receive array.



```
%DEFINE RADAR SENSOR PARAMETERS
transmitter = phased.Transmitter('PeakPower',0.001,'Gain',36);
receiver = phased.ReceiverPreamp('Gain',40,'NoiseFigure',4.5,...
           'SampleRate',fs);

tx_radiator = phased.Radiator('Sensor',tx_array,'OperatingFrequency',fc,...
    'PropagationSpeed',c,'WeightsInputPort',true);

rx_collector = phased.Collector('Sensor',rx_array,'OperatingFrequency',...
        fc,'PropagationSpeed',c);
```

Figure 4.4 Code to Define Radar Sensor Parameters

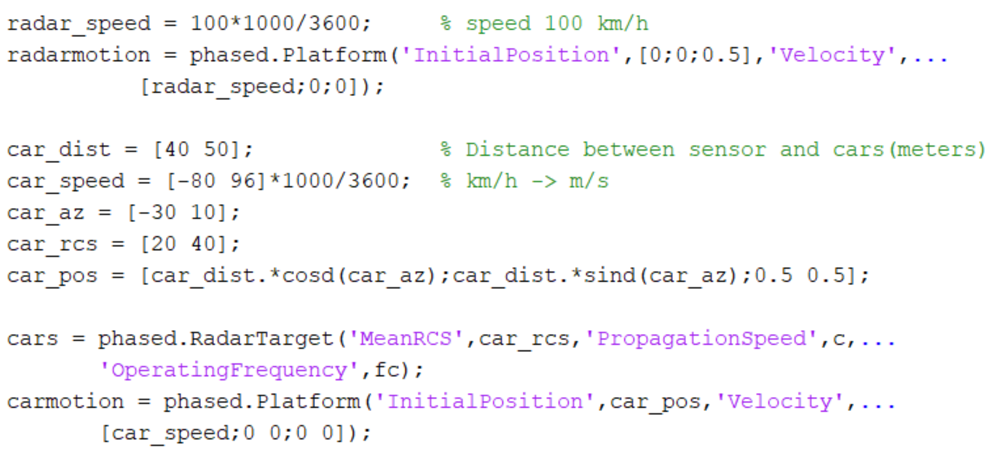
4.1.3 Radar Target Simulation

With the operating characteristics of the simulation defined, the targets that will be used in this simulation need to be defined. In this simulation, two moving cars are used as the targets. The motion of the radar is defined in Table 4.3; the car targets are defined by the parameters outlined in Table 4.4.

The cars are then created as a radar target using `phased.RadarTarget()`, and their motion is simulated using `phased.Platform()`. The command `phased.RadarTarget()` is used to define how the radar signal will be reflected from the radar targets [49].The MATLAB® command `phased.Platform()` models the translational motion of a platform in space [50]. This platform can be a target (airplane, vehicle, etc.) or a sonar or radar transmitter or receiver [50]. The model created assumes that the platform undergoes its motion at a constant velocity during the simulation; positions and velocities are defined within the global coordinate system [50].With the targets defined and their motion defined, the MIMO radar can now be simulated. The code used to define position and motion of the target is illustrated in Figure 4.5.

Table 4.3 Radar Motion Parameters

Radar Initial Position	x direction: 0 m
	y direction: 0 m
	Z direction: 0.5 m
Radar Velocity	x direction: 100 km/hr
	y direction: 0 km/hr
	Z direction: 0 km/hr



```
radar_speed = 100*1000/3600;     % speed 100 km/h
radarmotion = phased.Platform('InitialPosition',[0;0;0.5],'Velocity',...
          [radar_speed;0;0]);

car_dist = [40 50];              % Distance between sensor and cars(meters)
car_speed = [-80 96]*1000/3600;  % km/h -> m/s
car_az = [-30 10];
car_rcs = [20 40];
car_pos = [car_dist.*cosd(car_az);car_dist.*sind(car_az);0.5 0.5];

cars = phased.RadarTarget('MeanRCS',car_rcs,'PropagationSpeed',c,...
       'OperatingFrequency',fc);
carmotion = phased.Platform('InitialPosition',car_pos,'Velocity',...
       [car_speed;0 0;0 0]);
```

Figure 4.5 Code to Define Target Motion

Table 4.4 Car Target Parameters

Car 1	Velocity	-80 km/h
		y direction: 0 km/hr
		z direction: 0 km/hr
	Azimuth	-30 degrees
	RCS	20 m^2
	Starting Position	x direction: 34.64m
		y direction: -20m
		z direction: 0.5m
	Distance between car and autonomous vehicle sensor	40 m
Car 2	Velocity	x direction:96 km/h
		y direction: 0 km/hr
		z direction: 0 km/hr
	Azimuth	10 degrees
	RCS	40 m^2
	Starting Position	x direction: 49.24m
		y direction: 8.68m
		z direction: 0.5m
	Distance between car and autonomous vehicle sensor	50 m

4.1.4 MIMO Radar Simulation

The next step taken in this simulation is to create a radar simulation using the operating characteristics, radar specifications, and target parameters that were previously defined.

4.1.4.1 Waveform Creation

Before the radar simulation can be started, it is first required to choose and define the type of waveform that will be used. In this simulation, a Frequency Modulation Continuous Wave (FMCW) is used; FMCW is described in more detail in Section 2.1.1. The waveform used in this simulation is created using an embedded MATLAB® function, `helperDesignFMCWWaveform()`. The properties of this FMCW wave can be found in Table 4.5; a snapshot of the wave can be found in Figure 4.6 and Figure 4.7. The code used to create this waveform is found in Figure 4.8

Since this simulation uses operating characteristics that are based on vehicle ACC, a maximum range of 200 m and a range resolution of 1 m was used [51]. The range resolution of 1 m was chosen to ensure that targets which are separated by 1 m or greater are visible to the system. On a highway, it is not uncommon to have 1 m separation between cars (i.e. cars in different lanes). This is what necessitated the 1 m range resolution.

Generally, the sweep time for a FMCW wave should be approximately five to six times the round trip time to account for the time needed for the signal to travel the maximum unambiguous range [51]; in this thesis 5.5 was used. FMCW signals typically have a wide bandwidth. Setting the sample rate of the wave to twice the bandwidth (as per the Nyquist theorem) can stress hardware. Although this simulation is virtual, taking real world hardware limitations into account will allow for a more real-world result. Therefore, the sample rate of the FMCW signal can be set by considering the maximum beat frequency that the radar would need to detect; the sample rate of the signal would need to be, at a minimum, twice the maximum beat frequency. In practicality, the maximum speed of an average car is 230 km/hr [51]; therefore the maximum beat frequency would be

$$f_{b_{max}} = f_{r_{max}} + f_{d_{max}} \tag{4.1}$$

where $f_{r_{max}}$ is the beat frequency corresponding to the maximum range and $f_{d_{max}}$ is the maximum Doppler shift. In this system, $f_{r_{max}}$ and $f_{d_{max}}$ were found using

```
fr_max=
range2beat(range_max,sweep_slope,c)
```
(4.2)

```
fd_max = speed2dop(2*v_max,lambda)
```
(4.3)

It was found that $f_{r_{max}}$ was 27.2 MHz and $f_{d_{max}}$ was 127.9 Hz. Therefore,

$$\begin{aligned} f_{b_{max}} &= f_{r_{max}} + f_{d_{max}} \\ &= 27.254 \text{ MHz} \end{aligned} \tag{4.4}$$

In order to ensure that the Nyquist sampling theorem is respected, the sample rate of the wave is chosen based on whichever property is larger: twice the beat frequency or the sweep bandwidth. Since the sweep bandwidth of the signal is 150 MHz, the maximum beat frequency needs to be at least 300 MHz if we are to choose it. The following code was used to determine which was greater:

```
fs = max(2*fb_max,bw)
```
(4.5)

Therefore, the sweep bandwidth was found to be greater than the maximum beat frequency and was used as the sample rate. The signal that is created is an up-sweep linear FMCW signal, which is often referred to as a saw tooth [51]

Table 4.5 Transmitted FMCW Properties

Maximum Unambiguous Range	200 m
Sample Rate	150 MHz
Sweep Time	7.3 μs
Sweep Bandwidth	150 MHz

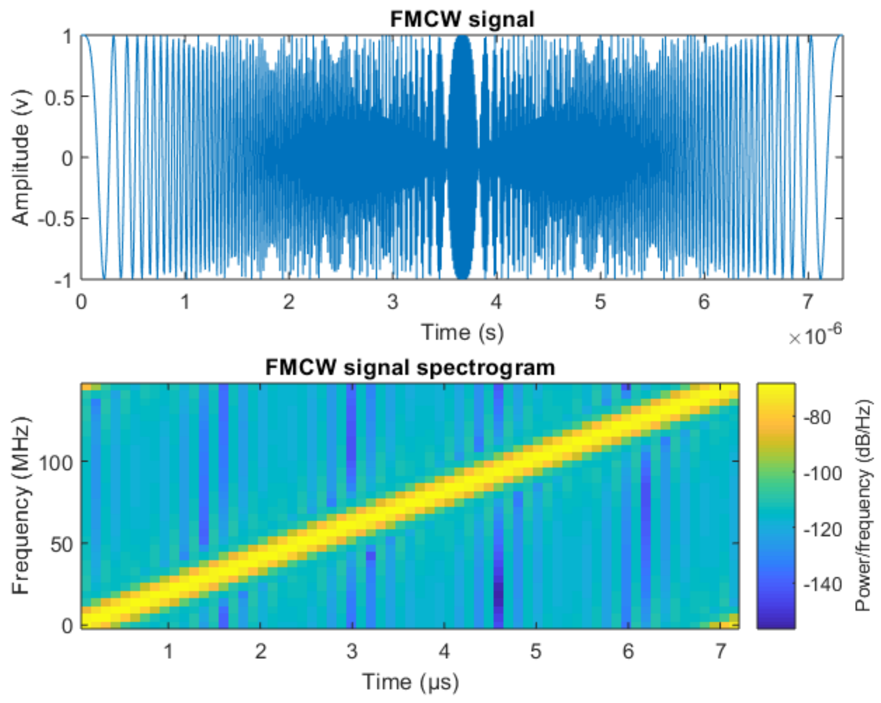


Figure 4.6 Transmitted FMCW Wave and Spectrogram

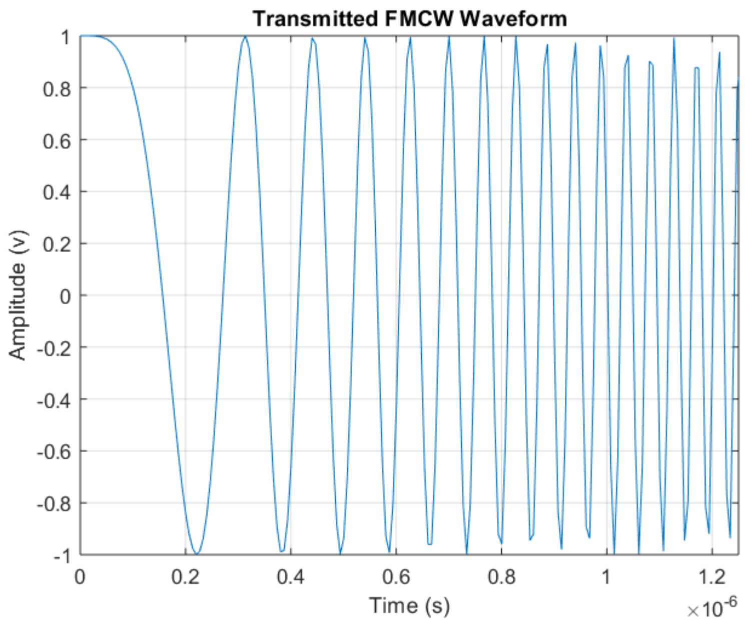
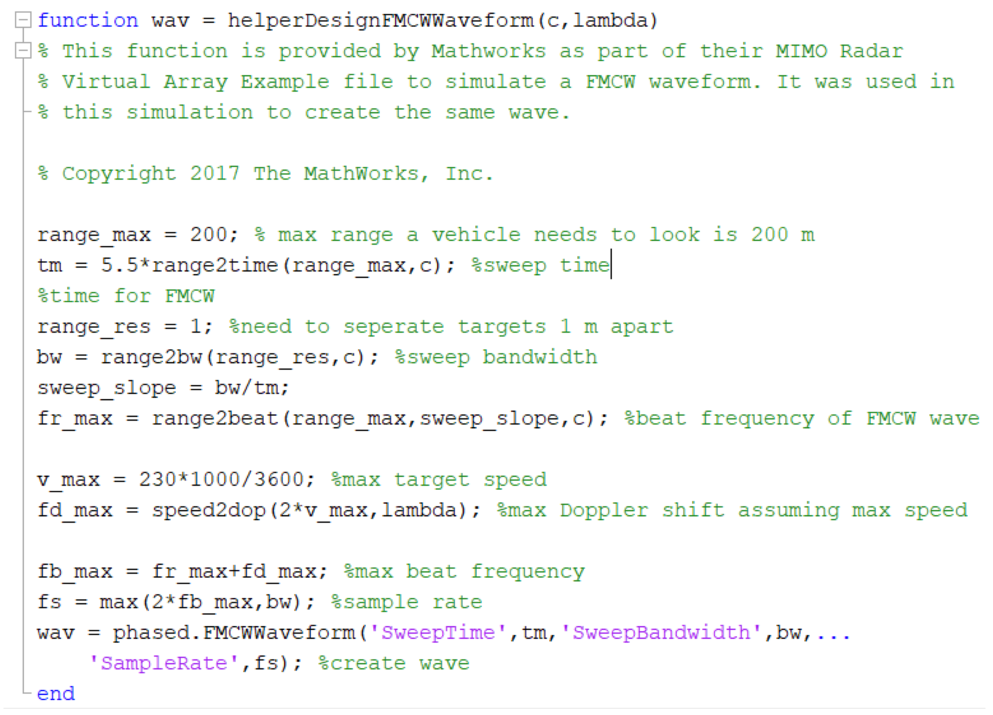


Figure 4.7 Transmitted FMCW Wave (zoomed in)



```
function wav = helperDesignFMCWWaveform(c,lambda)
% This function is provided by Mathworks as part of their MIMO Radar
% Virtual Array Example file to simulate a FMCW waveform. It was used in
% this simulation to create the same wave.

% Copyright 2017 The MathWorks, Inc.

range_max = 200; % max range a vehicle needs to look is 200 m
tm = 5.5*range2time(range_max,c); %sweep time
%time for FMCW
range_res = 1; %need to seperate targets 1 m apart
bw = range2bw(range_res,c); %sweep bandwidth
sweep_slope = bw/tm;
fr_max = range2beat(range_max,sweep_slope,c); %beat frequency of FMCW wave

v_max = 230*1000/3600; %max target speed
fd_max = speed2dop(2*v_max,lambda); %max Doppler shift assuming max speed

fb_max = fr_max+fd_max; %max beat frequency
fs = max(2*fb_max,bw); %sample rate
wav = phased.FMCWWaveform('SweepTime',tm,'SweepBandwidth',bw,...
    'SampleRate',fs); %create wave
end
```

Figure 4.8 Code to Create FMCW signal

4.1.4.2 Radar Execution

The radar function execution was carried out using a `for` loop that was designed to mimic the process of a physical radar emitting a signal and collecting returns that have been returned from a target. The `for` loop that was used to simulate MIMO radar operation is given in Appendix A. 2. Before the `for` loop is run, it is first necessary to define some characteristics of the loop, which are outlined in Table 4.6. The arrangement of elements is shown in Figure 4.9.

A decimation factor is used to decrease the sample rate; decreasing the sampling rate will make the `for` loop much less computationally intensive and reduce the time required to simulate the radar [52]. Therefore, since a decimation factor of two was used, the sampling frequency of the radar execution loop is now 4000 Hz since

$$f_{s_n} = \frac{f_s}{D} \tag{4.6}$$

where f_{s_n} is the new sampling frequency, f_s is the sampling frequency of the radar system (previous defined as 8000 Hz), and D is the decimation factor [46]. The number of sweeps dictates how many times the FMCW waveform will be emitted. Several sweeps are required to distinguish the Doppler shift of the moving targets; within one pulse, the Doppler frequency is indistinguishable from the beat frequency [51]. Therefore, 64 sweeps were chosen to be conducted. Because there are two transmit elements in the transmitting array, it is necessary to define which element will be radiating first; in this simulation element zero will transmit first.

There are six basic steps used to simulate the radar execution and to obtain the raw data cube that would be received by the physical receive array. These steps are summarized in Figure 4.11 and consist of updating radar and target position, transmitting the waveform, toggling the transmit element; propagating the signal and reflect off the target, dechirping the received radar return, and decimating the return (to reduce computation requirements). Dechirping is an operation in which the received signal and transmit signal are nixed together, which allows for the beat frequency to be measured [51]

The code used to simulate the radar operation is outlined in Figure 4.10. The radar and target position are updated at the beginning of each loop to ensure that the movement of both the radar and the target have been updated before the data is collected. During the first sweep, the radar and car motion are that of the starting position and velocities previously defined in Section 4.1.2 and Section 4.1.3. The signal is emitted using element 0 only and then the element is toggled to element 1 so that element 1 will emit the signal during the next sweep, Once the signal has been emitted, it is propagated through free space and reflected off the targets if they are present. The data is then extracted from the receive array, dechirped and decimated. To completely update the motion profile of the radar, the radar position, radar velocity, target position, target velocity, and the target angle of the cars needs to be updated at the start of each sweep.

Table 4.6 Radar Simulation Loop Operating Parameters

Decimation Factor	2
Number of Sweeps	64
Sampling Frequency	4000 Hz
Starting Transmit Element	Element 0

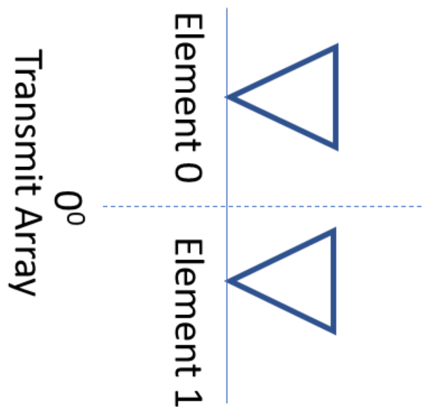
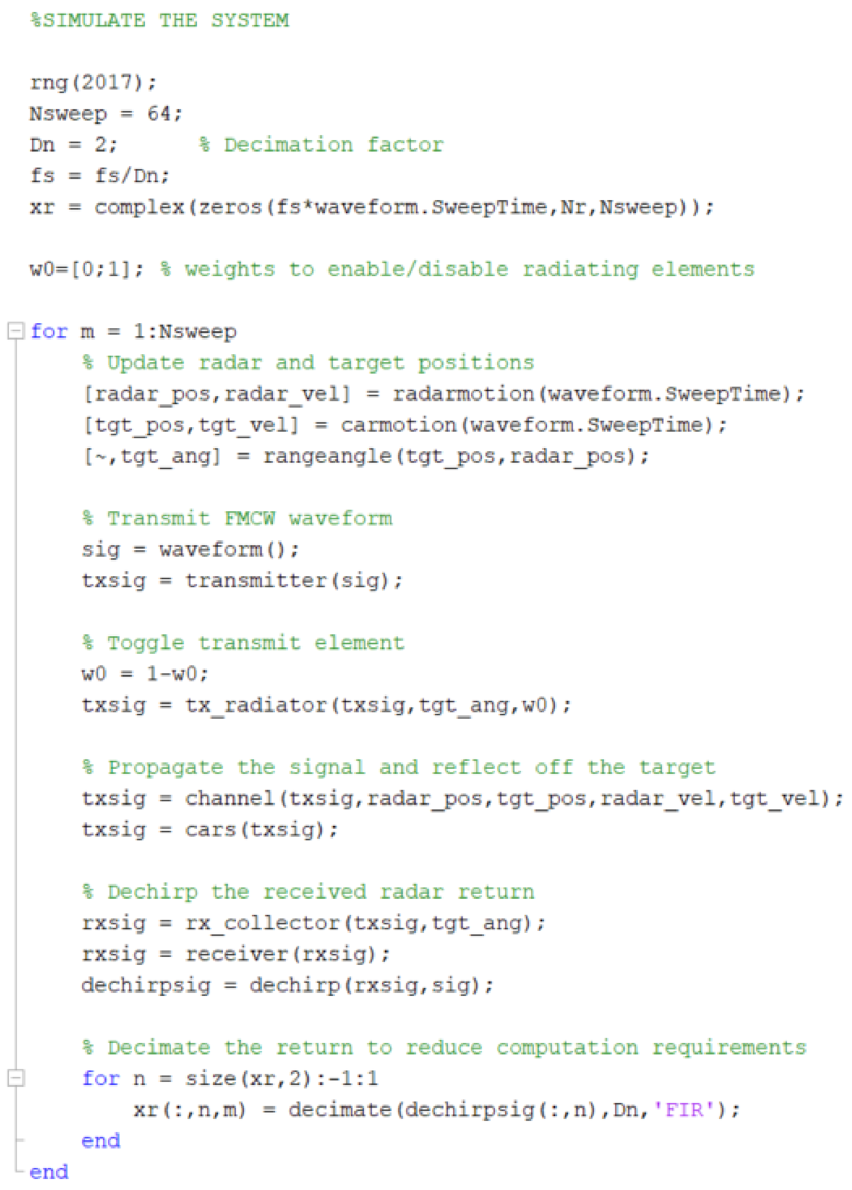


Figure 4.9 Transmit Array Element Arrangement



```matlab
%SIMULATE THE SYSTEM

rng(2017);
Nsweep = 64;
Dn = 2;        % Decimation factor
fs = fs/Dn;
xr = complex(zeros(fs*waveform.SweepTime,Nr,Nsweep));

w0=[0;1]; % weights to enable/disable radiating elements

for m = 1:Nsweep
    % Update radar and target positions
    [radar_pos,radar_vel] = radarmotion(waveform.SweepTime);
    [tgt_pos,tgt_vel] = carmotion(waveform.SweepTime);
    [~,tgt_ang] = rangeangle(tgt_pos,radar_pos);

    % Transmit FMCW waveform
    sig = waveform();
    txsig = transmitter(sig);

    % Toggle transmit element
    w0 = 1-w0;
    txsig = tx_radiator(txsig,tgt_ang,w0);

    % Propagate the signal and reflect off the target
    txsig = channel(txsig,radar_pos,tgt_pos,radar_vel,tgt_vel);
    txsig = cars(txsig);

    % Dechirp the received radar return
    rxsig = rx_collector(txsig,tgt_ang);
    rxsig = receiver(rxsig);
    dechirpsig = dechirp(rxsig,sig);

    % Decimate the return to reduce computation requirements
    for n = size(xr,2):-1:1
        xr(:,n,m) = decimate(dechirpsig(:,n),Dn,'FIR');
    end
end
```

Figure 4.10 Code used to Simulate Radar Operation

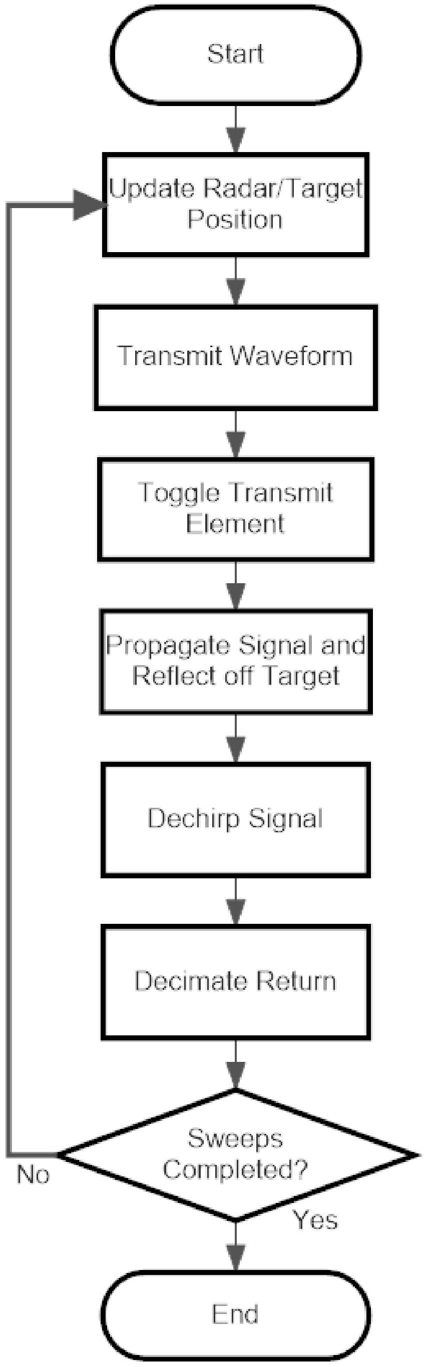


Figure 4.11 Radar Simulation Flowchart

4.1.4.3 Virtual Array Processing

The raw data that is received by the physical receive array must be processed in order to associate the returns with the correct transmit element. Since this simulation uses two transmit elements, the virtual array data can be processed by doing two sweeps of the data. This is done once the simulation described in Section 4.1.4.2 has been completed.

Since there were 64 radar sweeps completed, 32 of these sweeps will correspond to transmit element zero and 32 belong to transmit element one. Since the transmitters are toggled midway through the simulation, it is deduced that every other data cube will belong to the same transmitter, with the odd elements corresponding to element zero and the even corresponding to element One. This is counterintuitive to what one might think, however MATLAB® using an indexing base of one instead of zero. Therefore, to separate the data that is from different transmitters, the raw data is extracted and placed in an array to collocate the data from each transmitter together. These two arrays are then catenated together to create the virtual array data.

4.2 Results and Analysis

With the radar simulated and the virtual array created, the data is ready to be used with the MVDR algorithm. The algorithm that is used to carry out the calculation is described in detail in Section 3.2 and is used on both the CPU and GPU. The hardware setup used to run these simulations is the one outlined in Section 3.3.1.

The MATLAB® code found in Appendix A.2 was run using various combinations of transmitters and receivers in order to properly analyse its functionality. For simplicity, the MIMO radar system with only two transmit elements were simulated. Initially, the system was run with four receive elements to validate correct functionality of the MVDR algorithm. With the success of a four receive element system, the number of receivers was increased to eight, 16, 24, 48, 75, 100, and 300 to determine what effect this would have on the GPU speedup and execution times. All transmit and receive arrays in the simulated systems used 0.5λ spacing. In a real-world scenario, an ACC system would not have 24, 48, 75, 100 or 300 receiving elements, however in this scenario these numbers are used to determine how an increased number of receivers will affect the speedups and execution times.

The CPU and GPU MVDR spectrum when two transmitters and four receivers are used can be found in Figure 4.12 and Figure 4.13; the two way beam pattern for the system can be seen in Figure 4.14. From these figures, it is deduced that the MIMO radar simulation that was conducted, the virtual array construction, and the MVDR algorithm is functioning correctly. This was determined since the spectrum clearly shows two independent targets in the spectrum; the targets found in the spectrum correctly represent the azimuth of both cars.

For a two transmitter, four receiver system, it was determined that the overall MSE was 0.0167 and the root MSE was 0.1294. The MSE for all angles within the spectrum can be seen in Figure 4.15. Therefore, both data sets are shown to be similar in value, as expected based on the observed similarity shown in Figure 4.12 and Figure 4.13. Variations around the MSE are seen to be greatest around the angles at which the targets occur, -30 and 10 degrees. This is not unsurprising as this would be where the greatest variation in calculation would occur. The small MSE and root MSE also confirms the designed MVDR algorithm’s correct functionality on the GPU, and that a GPU can be integrated with a MIMO radar system.

The data obtained from the simulations run using the MIMO radar system is summarized in Table 4.7 and Figure 4.16. Table 4.7 details the execution time for the CPU and GPU execution time for a varying number of receiving elements, as well as the speed up that was obtained by utilizing the GPU; Figure 4.16 shows the CPU and GPU execution times pictographically. From this table and this figure, it is shown that the GPU provides a significant benefit when the complete MIMO radar setup is simulated- all scenarios showed a speedup achieved when using the GPU. The speedups obtained were 5.04x, 4.97x, 5.69x, 6.26x, 6.00x, 5.96x, 7.26x, and 7.71x.

The execution speedup highlighted in Table 4.7 shows that as the number of receivers is increased the overall speedup obtained also increases. This is an expected result, since large data sets will take longer to execute serially than in a parallel operation. The strength of the GPU's parallel computing capability is highlighted once the receivers are increased to numbers in the hundreds (i.e. 100 and 300), as the overall speed up of the MVDR algorithm is increased to 7.26x and 7.71x. It can be extrapolated that the overall speedup in the system would continue to increase as the number of receivers in the system also increases. The system in this thesis is more complex than a ULA system, but not as complex as those commonly used in MIMO communication systems or much larger MIMO radar systems (tens/hundreds of transmitters and/or receivers). It is expected that as the number of transmitters and receivers were to increase, the efficiency obtained in using a GPU would be much greater than those obtained in this thesis.

There are limitations that come with using MATLAB® as the programming platform. While MATLAB® does make it much less time intensive to program parallel code compared to CUDA, there is a design trade off that occurs- the programmer does not have the ability to specify memory location or dictate which type of memory will be used for the mathematical operations. Since shared memory is much faster than global, it can be inferred that if the programmer was able to exploit shared memory for the MVDR algorithm, the overall speedup would be greater. The faster execution time associated with shared memory is a well-documented property and once that programmers are advised to exploit [2], [28]. In this simulation, the speedup obtained is still evident, however with a bigger, more complex system (thousands/millions of transmit/received elements instead of the hundreds in this system at its greatest) utilizing shared memory would allow for an even greater advantage. Although the storage space available for shared memory is small (kilobytes on average), the memory access time is small and would require many less reads and writes from global memory. A read/write would be required from global into shared to utilize shared memory and then a single write would be needed to transfer the data back to global. If global memory is strictly used, many more read/writes are required to manipulate the data and then store the result. The grid size, block size, and number of threads also cannot be dictated at run time. Depending on the data and computations to be carried out, changing the grid size, block size, and number of executed threads could also decrease execution time. Unfortunately, neither of these possibilities were unable to be explored due to MATLAB®'s limitations.

Overall, this simulation was a success and has proven that a simulated MIMO radar system can be successfully integrated with a GPU. This simulation was completely software based; it shows a promising result towards practical applications with hardware and a physical radar setup.

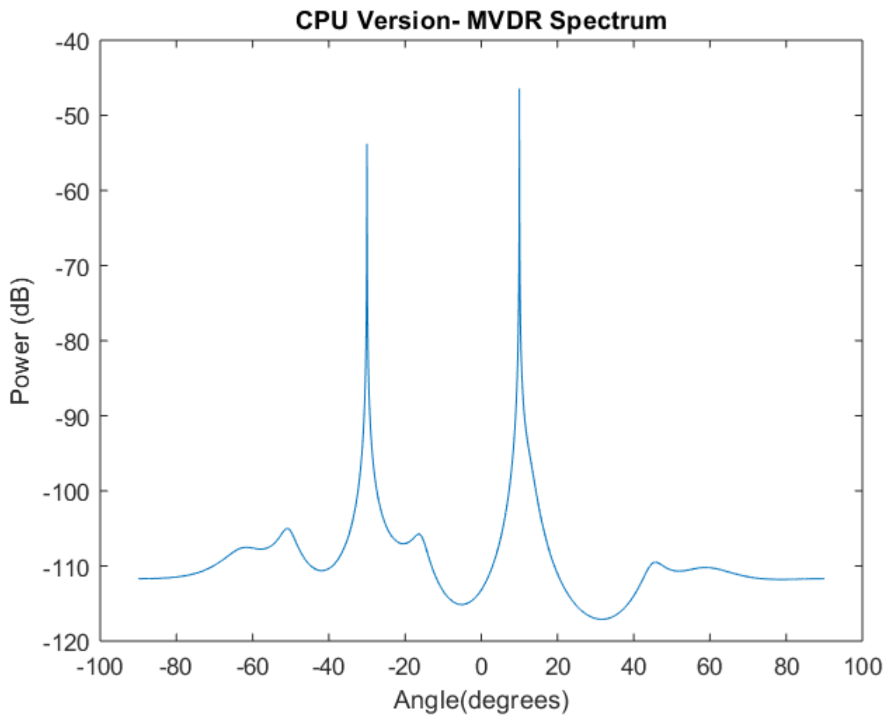


Figure 4.12 CPU MVDR Spectrum for two transmitting elements and four receiving elements

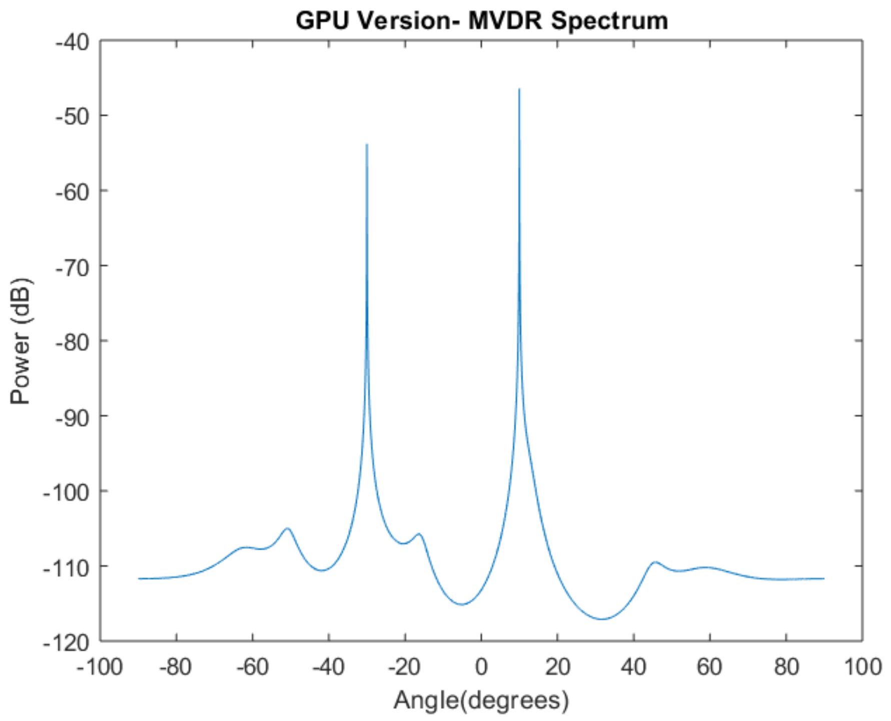


Figure 4.13 GPU MVDR Spectrum for two transmitting elements and four receiving elements

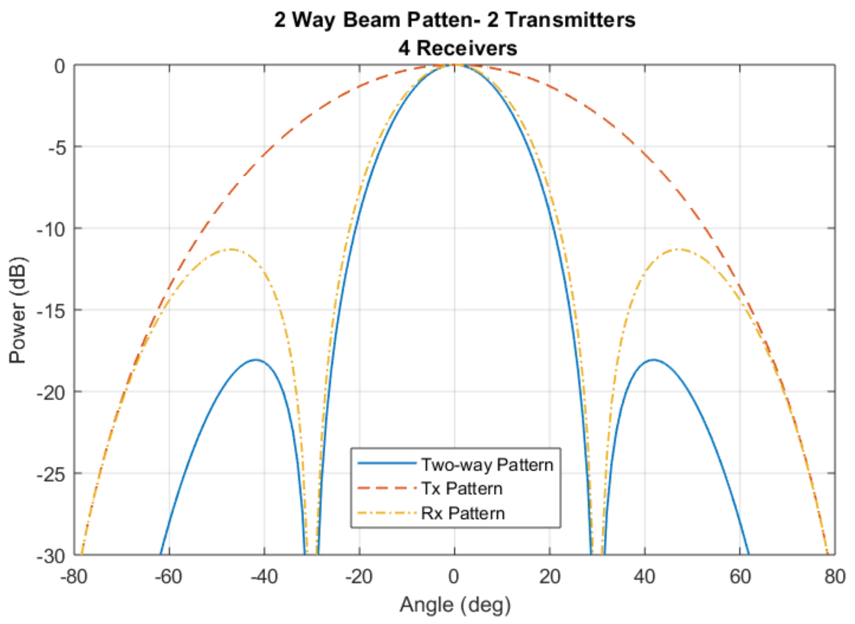


Figure 4.14 Two Way Beam Pattern for Four Receivers

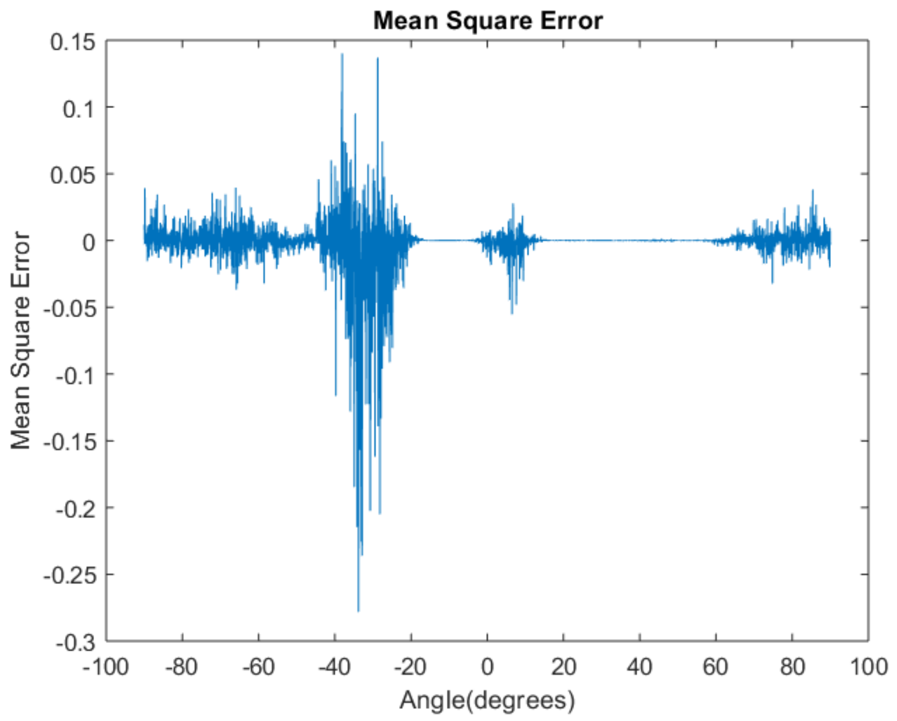


Figure 4.15 MSE for Two Transmit Elements and Four Receive Elements

Table 4.7 CPU and GPU Execution Time and Speedup for Two Transmit Elements

Number of Receivers	**CPU Execution Time (ms)**	**GPU Execution Time (ms)**	**Speedup**
4	269.68	53.57	5.04
8	270.68	54.46	4.97
16	343.704	60.44	5.69
24	387.34	61.92	6.26
48	529.22	88.30	6.00
75	703.60	118.13	5.96
100	890.50	122.78	7.26
300	2427.50	314.66	7.71

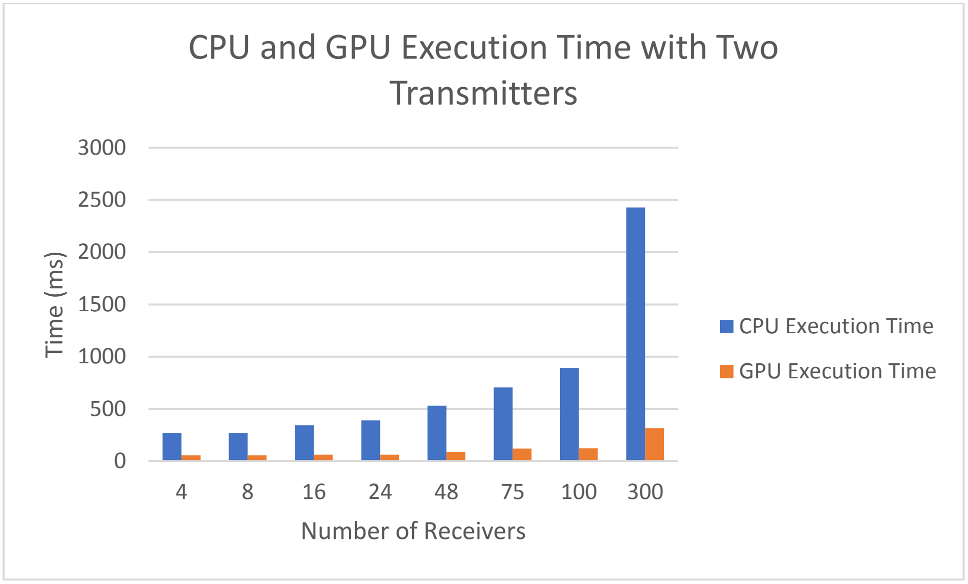


Figure 4.16 CPU vs GPU Execution Time for Two Transmitters

5 Conclusion

5.1 Summary

As discussed in Chapter 2, a MIMO radar can transmit and receive multiple signals at once. Once the return signals have been collected by the receiver, extensive signal processing is required to determine if a target has been detected. A MIMO radar offers many advantages; however, it is computationally intensive. Historically, this signal processing has been done using CPUs. There is much room left to explore the GPU's computing capabilities and applications to current algorithms. A GPU contains thousands of cores and can be programmed to execute parallel code which decreases execution time for computationally intensive programs in comparison to a CPU. Most of the research papers and documentation read and analysed over the course of this thesis found little research has been done into the use of GPUs with MIMO radar systems. Most of the research that has been published uses GPU with traditional SISO radar or MIMO in a communications application. The lack of integration of a GPU with a MIMO radar is a significant gap that should be more thoroughly explored.

Chapter 3 outlined the design of a ULA scenario which was used to determine if the developed MVDR algorithm was functioning correctly. Its simulation environment was explained, and specialty toolboxes required for the simulation were examined. The results for this simulation were presented, along with the hardware used to run the simulations. It was found that the MVDR algorithm's functionality was as desired, as it was able to detect the four targets used. While this design showed a speedup when the GPU was used, it is not recommended to combine ULAs and GPUs in practice, considering the simplicity of the system and the complexity involved with parallel programming.

Chapter 4 described the design of the simulated MIMO radar and its results. In this chapter, a MIMO radar is simulated based on a radar system that would be embedded on a moving car for ACC; two moving cars are used as targets. The results that were obtained are analysed, and it was found that GPUs can successfully be used to speedup beamforming calculations for a MIMO radar.

5.2 Conclusions

The hypothesis for this thesis was that it would be possible to design and develop a serial and parallel MVDR algorithm that would carry out target detection for a MIMO radar system. The designed MVDR algorithm was tested using two different systems, a ULA system and a MIMO radar system.

The designed MVDR algorithm was first tested within a ULA since the system is simple and allows for easy debugging of the MVDR algorithm. The goal of this test environment was to validate the functionality of the MVDR algorithm; decreasing execution time was not a priority for this system. The initial debugging of the algorithm found that two of MATLAB®'s built in functions, `cov()` and `inv()` did not function correctly on the GPU. Unfortunately, due to MATLAB®'s high-level programming language and inability to access source code meant that the exact cause of incorrect functionality could be verified. The equation to calculate the MVDR power spectrum was changed and used matrix diagonalization. Using diagonalization presents a limitation to the algorithm, as it will not function correctly if there is interference present within the system. With this change, the MVDR algorithm was able to resolve all four targets which were present within the system. It was also found that this system provided different speedups for different number of elements within the ULA: 1.11x for eight elements, 2.70x for 16 elements, 1.78x for 32 elements, 1.87 for 64 elements, and 1.79x for 128 elements. This scenario confirms what was initially suspected, that it is possible to parallelize the MVDR algorithm and use it for successful target detection.

Once the designed MVDR algorithm was tested, it was then implemented within a MIMO radar system. The system that was designed used an ACC system on board an autonomous car as the basis for all radar and target parameters; the waveform used for the transmitted was an FMCW wave that used TDM to achieve orthogonality. This system was simulated using two transmitters and four, eight, 16, 24, 48, 75, 100, and 300 receivers. All these combinations were successful at resolving the two moving targets that were within the system and produced speedups of 5.04x, 4.97x, 5.69x, 6.26x, 6.00x, 5.96x, 7.26x, and 7.71x respectively. This simulation proved that not only can the MVDR algorithm be parallelized and produce a system speedup, it is also possible to integrate a GPU within a MIMO radar system. In conclusion, a GPU can reduce the time required to carry out the MVDR beamforming algorithm, as demonstrated in the ULA and MIMO radar system simulations. While MATLAB® allows for an easier coding experience, it does provide the programmer with limitations that could impact the observed overall speedup. If achieving the fastest possible execution time is of highest importance, the programmer should opt to use CUDA or MEX files to have the greatest possible influence over programming; if ease of programming is a higher priority, then MATLAB® is an acceptable software program to use.

5.3 Contributions

The most important contributions of this work are:

a) the design and implementation of a serial and parallel version of the MVDR algorithm;
b) the implementation of a MIMO radar system coded in MATLAB® using the developed MVDR algorithm to accurately detect both stationary and moving targets; and
c) validation that a GPU can be successfully integrated into a MIMO radar simulation.

As a result of the work conducted in this thesis, it has been proven that a simulated MIMO radar system can be successfully integrated with a GPU, and that the GPU will reduce the execution time required to carry out beamforming.

5.4 Future Work

There are many areas for research in MIMO radar and GPUs. Modifying the simulation run in Section 3.3 to allow for different input waveforms should be implemented. This change would allow for the study of how different waveforms impact GPU performance and the overall execution time of the system.

The equation used to carry out the MVDR algorithm should be investigated further to find a solution which does not involve matrix diagonalization since this limits the systems that the designed MVDR algorithm can be integrated in. It should also be investigated further as to the exact reason why well tested MATLAB® commands, such as `cov()` and `inv()` have incorrect functionality on a GPU and ways to mitigate this issue.

The code in Section 3.3 should be made adaptive to allow for an increase beyond two transmitting elements; different targets should also be used. With more transmitting elements and various targets (stationary and moving) will provide a more holistic picture of the benefits of using GPUs in MIMO radar processing.

Lastly, the designed algorithm and a GPU should be combined with a physical MIMO radar. Moving the developed algorithm from a purely software simulation would increase its robustness and allow for real world application validation.

Bibliography

[1] M. Wu, Y. Sun, and J. R. Cavallaro, “Reconfigurable Real-time MIMO Detector on GPU,” presented at the 43rd Asilomar Conf. Signals, Systems, and Computers, Pacific Grove, CA, USA, 2009, pp. 690–694.

[2] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*, Third. Cambridge, MA, United State: Elsevier, 2017.

[3] J. Bathurst, “Multiple Input Multiple Output (MIMO) radar detection of moving targets on the ocean surface,” MASc Thesis, Division of Graduate Studies, Royal Military College of Canada, Kingston, ON, 2017.

[4] R. S. Perdana, B. Sitohang, and A. B. Suksmono, “A survey of graphics processing unit (GPU) utilization for radar signal and data processing system,” in *2017 6th International Conference on Electrical Engineering and Informatics (ICEEI)*, Langkawi, 2017, pp. 1–6.

[5] Y. Liu, X. Wan, and X. Sun, “GPU parallel acceleration of target detection in passive radar system,” in *2016 CIE International Conference on Radar (RADAR)*, Guangzhou, China, 2016, pp. 1–4.

[6] T. Nylanden, J. Janhunen, O. Silven, and M. Juntti, “A GPU implementation for two MIMO-OFDM detectors,” in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, Samos, Greece, 2010, pp. 293–300.

[7] C. M. Jozsa, F. Domene, G. Pinero, A. Gonzalez, and A. M. Vidal, “Efficient GPU implementation of Lattice-Reduction-Aided Multiuser Precoding,” in *10th ISWCS*, Ilmenau, Germany, 2013, pp. 1–5.

[8] T. Chen and H. Leib, “GPU acceleration for fixed complexity sphere decoder in large MIMO uplink systems,” in *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, Halifax, NS, Canada, 2015, pp. 771–777.

[9] Y. Sun and J. R. Cavallaro, “High throughput VLSI architecture for soft-output mimo detection based on a greedy graph algorithm,” in *Proceedings of the 19th ACM Great Lakes symposium on VLSI - GLSVLSI ’09*, Boston Area, MA, USA, 2009, p. 445.

[10] S. Roger, C. Ramiro, A. Gonzalez, V. Almenar, and A. M. Vidal, “Fully Parallel GPU Implementation of a Fixed-Complexity Soft-Output MIMO Detector,” *IEEE Transactions on Vehicular Technology*, vol. 61, no. 8, pp. 3796–3800, Oct. 2012.

[11] J. Li and P. Stoica, “MIMO Radar: Concepts, Performance, Enhancements, and Applications,” in *MIMO Radar Signal Processing*, 1st ed., Hoboken, NJ, United States: John Wiley and Sons, 2009, pp. 65–71.

[12] F. Jameel, Faisal, M. A. A. Haider, and A. A. Butt, “Massive MIMO: A survey of recent advances, research issues and future directions,” in *2017 International Symposium on Recent Advances in Electrical Engineering (RAEE)*, Islamabad, 2017, pp. 1–6.

[13] J. J. M. de Wit, W. L. van Rossum, and A. J. de Jong, “Orthogonal waveforms for FMCW MIMO radar,” in *2011 IEEE RadarCon (RADAR)*, 2011, pp. 686–691.

[14] I. Bekkerman and J. Tabrikian, “Target Detection and Localization Using MIMO Radars and Sonars,” *IEEE Transactions on Signal Processing*, vol. 54, no. 10, pp. 3873–3883, Oct. 2006.

[15] J. O. Hinz and U. Zolzer, “A MIMO FMCW Radar Approach to HFSWR,” *Advances Radio Science*, vol. 9, pp. 159–163.

[16] A. Ganis *et al.*, “A Portable 3-D Imaging FMCW MIMO Radar Demonstrator With a 24×24 Antenna Array for Medium-Range Applications,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 56, no. 1, pp. 298–312, Jan. 2018.

[17] Z. Fang, L. Lou, C. Yang, K. Tang, and Y. Zheng, “A Ku-band FMCW Radar on Chip for Wireless Micro Physiological Signal Monitoring by Interferometry Phase Analysis,” in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–4.

[18] P. Sévigny, P. W. Moo, and T. Laneve, “Experimental Verification of Multiple-input Multiple Output (MIMO) Beamforming Capabilities Using a

Time-division Coherent MIMO Radar,” *IET Radar Sonar Navig.*, p. 62, Aug. 2015.

[19] E. Brookner, “MIMO radars demystified — and their conventional equivalents,” in *2016 IEEE International Symposium on Phased Array Systems and Technology (PAST)*, Waltham, MA, USA, 2016, pp. 1–10.

[20] E. Fishler, A. Haimovich, R. Blum, D. Chizhik, L. Cimini, and R. Valenzuela, “MIMO radar: an idea whose time has come,” in *Proceedings of the 2004 IEEE Radar Conference (IEEE Cat. No.04CH37509)*, Philadelphia, PA, USA, 2004, pp. 71–78.

[21] S. A. Vorobyov, “Principles of minimum variance robust adaptive beamforming design,” *Signal Processing*, vol. 93, no. 12, pp. 3264–3277, Dec. 2013.

[22] J. Capon, “High- Resolution Frequency-Wavenumber Spectrum Analysis,” *Proceedings of the IEEE*, vol. 57, no. 57, No. 8, pp. 1408–1418, Aug. 1969.

[23] J. Sanson, A. Gameiro, D. Castanheira, and P. P. Monteiro, “Comparison of DoA Algorithms for MIMO OFDM Radar,” in *2018 15th European Radar Conference (EuRAD)*, 2018, pp. 226–229.

[24] J. Li and P. Stoica, Eds., *Robust adaptive beamforming / edited by Jian Li and Petre Stoica*. Hoboken, NJ: John Wiley, 2006.

[25] S. Ahmed and M.-S. Alouini, “Low complexity receiver design for MIMO-radar,” in *2012 IEEE Globecom Workshops*, Anaheim, CA, USA, 2012, pp. 1394–1398.

[26] Y. Zhang and J. Wang, “Transmit-receive beamforming for MIMO radar,” in *2010 2nd International Conference on Signal Processing Systems*, 2010, vol. 3, pp. V3-803-V3-806.

[27] Y. Boers and J. N. Driessen, “Multitarget particle filter track before detect application,” *IEE Proceedings - Radar, Sonar and Navigation*, vol. 151, no. 6, p. 351, 2004.

[28] NVIDIA, *NVIDIA CUDA C Programming Guide*. Santa Clara, CA, 2012.

[29] The MathWorks, Inc., “Parallel Computing Toolbox User’s Guide.” 2019.

[30] “Accessing Advanced CUDA Features Using MEX - MATLAB & Simulink Example.” [Online]. Available: https://www.mathworks.com/help/parallel-

computing/examples/accessing-advanced-cuda-features-using-mex.html. [Accessed: 13-Aug-2019].

[31] "Introducing MEX Files - MATLAB & Simulink." [Online]. Available: https://www.mathworks.com/help/matlab/matlab_external/introducing-mex-files.html. [Accessed: 13-Aug-2019].

[32] "What You Need to Build MEX Files - MATLAB & Simulink." [Online]. Available: https://www.mathworks.com/help/matlab/matlab_external/what-you-need-to-build-mex-files.html. [Accessed: 14-Aug-2019].

[33] The MathWorks, Inc., "Phased Array System Toolbox Getting Started Guide." .

[34] "Uniform linear array - MATLAB." [Online]. Available: https://www.mathworks.com/help/phased/ref/phased.ula-system-object.html#bso1s4m-2. [Accessed: 08-Jul-2019].

[35] "Simulate received plane waves - MATLAB." [Online]. Available: https://www.mathworks.com/help/phased/ref/phased.ula.collectplanewave.html. [Accessed: 19-Jun-2019].

[36] "Minimum variance distortionless response (MVDR) beamformer weights - MATLAB mvdrweights." [Online]. Available: https://www.mathworks.com/help/phased/ref/mvdrweights.html. [Accessed: 19-Jun-2019].

[37] "Covariance - MATLAB cov." [Online]. Available: https://www.mathworks.com/help/matlab/ref/cov.html. [Accessed: 21-Sep-2019].

[38] "Matrix inverse - MATLAB inv." [Online]. Available: https://www.mathworks.com/help/matlab/ref/inv.html?s_tid=doc_ta. [Accessed: 20-Sep-2019].

[39] "LU matrix factorization - MATLAB lu." [Online]. Available: https://www.mathworks.com/help/matlab/ref/lu.html?searchHighlight=lu%20decomposition&s_tid=doc_srchtitle. [Accessed: 20-Sep-2019].

[40] J. Benesty, J. Chen, and Y. Huang, "A Generalized MVDR Spectrum," *IEEE Signal Processing Letters*, vol. 12, no. 12, Dec. 2005.

[41] Y. Xiao, J. Yin, H. Qi, H. Yin, and G. Hua, “MVDR Algorithm Based on Estimated Diagonal Loading for Beamforming,” *Mathematical Problems in Engineering*, 2017.

[42] “Solve systems of linear equations Ax = B for x - MATLAB mldivide \.” [Online]. Available: https://www.mathworks.com/help/matlab/ref/mldivide.html. [Accessed: 20-Sep-2019].

[43] “GeForce GTX TITAN | Specifications | GeForce.” [Online]. Available: https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications. [Accessed: 13-Jun-2019].

[44] “CUDA GPUs,” *NVIDIA Developer*, 04-Jun-2012. [Online]. Available: https://developer.nvidia.com/cuda-gpus. [Accessed: 14-Jun-2019].

[45] NVIDIA, “NVIDIA Kepler GK110 GK210 Architecture White Paper.” .

[46] “Increasing Angular Resolution with MIMO Radars - MATLAB & Simulink.” [Online]. Available: https://www.mathworks.com/help/phased/examples/increasing-angular-resolution-with-mimo-radars.html. [Accessed: 21-Aug-2019].

[47] “Adaptive cruise control,” *Wikipedia*. 05-Aug-2019.

[48] N. Pandey, “Beamforming in MIMO Radar,” National Institute of Technology Rourkela, Rourkela, India, 2014.

[49] “Radar target - MATLAB.” [Online]. Available: https://www.mathworks.com/help/phased/ref/phased.radartarget-system-object.html?searchHighlight=phased.RadarTarget&s_tid=doc_srchtitle. [Accessed: 21-Aug-2019].

[50] “Model platform motion - MATLAB.” [Online]. Available: https://www.mathworks.com/help/phased/ref/phased.platform-system-object.html?searchHighlight=phased.Platform&s_tid=doc_srchtitle. [Accessed: 21-Aug-2019].

[51] “Automotive Adaptive Cruise Control Using FMCW Technology - MATLAB & Simulink.” [Online]. Available: https://www.mathworks.com/help/phased/examples/automotive-adaptive-cruise-control-using-fmcw-technology.html. [Accessed: 21-Aug-2019].

[52] G. J. Dolecek and J. C. Suarez, “Improving alias rejection in comb decimation filters for odd decimation factors,” in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2017, pp. 397–400.

A MATLAB® Code

A.1 ULA Simulation Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%
%
%This file creates a ULA simulation with CPU/GPU
implementation of
%MVDR beamforming.
%
% This file is structured in the following format
%   a. Define signals and target directions
%   b. Create the ULA and simulate the system using
plane waves
%   c. CPU version of MVDR beamforming is carried out
%   d. GPU Version of MVDR beamforming is carried out
%
%NOTE: The MVDR beamforming used in this file is hand
coded for easier
%comparison between CPU/GPU. This version uses the
diagonalization of a
%matrix instead of determining a matrix inverse to
avoid the issues that
%arise with unstable inversion.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%

clearvars;
close all;
clear all;

%sampling frequency
```

```
fs = 8000;
t = (0:1/fs:1).';

%signals
x1 = cos(2*pi*t*100);
x2 = cos(2*pi*t*200);
x3 = cos(2*pi*t*300);
x4 = cos(2*pi*t*600);

sig_dir = [-40 -20 10 30]
N=256; %number of elements in the array
fc = 300.0e6;
c = physconst('LightSpeed');
lambda = c/fc;

%create array
array =
phased.ULA('NumElements',N,'ElementSpacing',lambda/2);

%simulate received signals using signals above
x = collectPlaneWave(array,[x1 x2 x3 x4],sig_dir,fc);

%%CPU MVDR COMPUTATION

% Compute the MVDR spatial spectrum
steervec = phased.SteeringVector('SensorArray',array);
teta=linspace(-45,45,1000);
sv = steervec(fc,teta);
Rxx=transpose(x)*x;

%start the MVDR algorithm and record execution time
tic;
S=diag(transpose(sv)*(Rxx\sv));
toc;
S_dB=-10*log10(S);

%plot specturm
figure;
plot(teta,S_dB);
title('CPU Version- MVDR Spectrum');
xlabel('Angle(degrees)');
ylabel('Power (dB)');
```

```
%%GPU MVDR COMPUTATION

%transfer required data to GPU
sv_parallel = gpuArray(sv);
Rxx_parallel = gpuArray(Rxx);

%carry out MVDR algorithm on GPU
tic;
S_parallel=diag(transpose(sv_parallel)*(Rxx_parallel\sv
_parallel));
toc;

%transfer data back to Matlab workspace
mvdr_spectrum_gpu = gather(S_parallel);
mvdr_spectrum_gpu_dB=-
10*log10(abs((mvdr_spectrum_gpu)));

%plot specturm
figure;
plot(teta,mvdr_spectrum_gpu_dB);
title('GPU Version- MVDR Spectrum');
xlabel('Angle(degrees)');
ylabel('Power (dB)');
```

A.2 MIMO Radar Simulation Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%
%
%This file creates a MIMO Radar simulation with CPU/GPU
implementation of
%MVDR beamforming.
%
% This file is structured in the following format
%   a. operating characteristics of the radar are set
%   b. transmitting and receiving arrays of the radar
are created
%   c. CPU version of MVDR beamforming is carried out
%   d. GPU Version of MVDR beamforming is carried out
%
```

```
%Radar operation: This file simulates a TDM MIMO radar.
The TX array will
%"transmit" a signal which will then bounce off a
target and be received
%by the RX array. This data is then used to construct
the virtual array,
%which then carries out the MVDR algorithm that was
validated in the .m
%file MIMO_MVDR_matrix_diagonalization.m
%
%NOTE: The MVDR beamforming used in this file is hand
coded for easier
%comparison between CPU/GPU. This version uses the
diagonalization of a
%matrix instead of determining a matrix inverse to
avoid the issues that
%arise with unstable inversion.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%

close all;
clear all;


%DEFINE OPERATING PARAMETERS
fs = 8000;
fc = 300.0e6;
t = (0:1/fs:1).';
c = physconst('LightSpeed');
lambda =c/fc;
teta = linspace(-90,90,5000);

waveform = helperDesignFMCWWaveform(c,lambda);
fs = waveform.SampleRate;

%CREATE TX AND RX ARRAYS

%number of transmitters
Nt=2;

%number of receivers
Nr=4;
```

```
%create array spacing (2lamda for Tx, 0.5lamda for dr)
% dt = Nr*lambda/2;
dt =lambda/2;
dr = lambda/2;

%create arrays
tx_array = phased.ULA(Nt,dt);
rx_array = phased.ULA(Nr,dr);
virtual_array = phased.ULA(Nt*Nr, dr);


%DEFINE RADAR SENSOR PARAMETERS
transmitter =
phased.Transmitter('PeakPower',0.001,'Gain',36);
receiver =
phased.ReceiverPreamp('Gain',40,'NoiseFigure',4.5,'Samp
leRate',fs);

tx_radiator =
phased.Radiator('Sensor',tx_array,'OperatingFrequency',
fc,...
    'PropagationSpeed',c,'WeightsInputPort',true);

rx_collector =
phased.Collector('Sensor',rx_array,'OperatingFrequency'
,fc,...
    'PropagationSpeed',c);


%DEFINE POSITION AND MOTION OF TARGET

%target used in this simulation is an ego vehicle and 2
cars

radar_speed = 100*1000/3600;     % speed 100 km/h
radarmotion =
phased.Platform('InitialPosition',[0;0;0.5],'Velocity',
[radar_speed;0;0]);

car_dist = [40 50];              % Distance between
sensor and cars (meters)
car_speed = [-80 96]*1000/3600;  % km/h -> m/s
```

```
car_az = [-30 10];
car_rcs = [20 40];
car_pos =
[car_dist.*cosd(car_az);car_dist.*sind(car_az);0.5
0.5];

cars =
phased.RadarTarget('MeanRCS',car_rcs,'PropagationSpeed'
,c,'OperatingFrequency',fc);
carmotion =
phased.Platform('InitialPosition',car_pos,'Velocity',[c
ar_speed;0 0;0 0]);

%DEFINE PROPOGATION MODEL

%assumedto be free space
channel = phased.FreeSpace('PropagationSpeed',c,...

'OperatingFrequency',fc,'SampleRate',fs,'TwoWayPropagat
ion',true);

%SIMULATE THE SYSTEM

rng(2017);
Nsweep = 64;
Dn = 2;      % Decimation factor
fs = fs/Dn;
xr = complex(zeros(fs*waveform.SweepTime,Nr,Nsweep));

w0=[0;1]; % weights to enable/disable radiating
elements

for m = 1:Nsweep
    % Update radar and target positions
    [radar_pos,radar_vel] =
radarmotion(waveform.SweepTime);
    [tgt_pos,tgt_vel] = carmotion(waveform.SweepTime);
    [~,tgt_ang] = rangeangle(tgt_pos,radar_pos);

    % Transmit FMCW waveform
    sig = waveform();
    txsig = transmitter(sig);
    % Toggle transmit element
    w0 = 1-w0;
    txsig = tx_radiator(txsig,tgt_ang,w0);

    % Propagate the signal and reflect off the target
    txsig = 
channel(txsig,radar_pos,tgt_pos,radar_vel,tgt_vel);
    txsig = cars(txsig);

    % Dechirp the received radar return
    rxsig = rx_collector(txsig,tgt_ang);
    rxsig = receiver(rxsig);
    dechirpsig = dechirp(rxsig,sig);

    % Decimate the return to reduce computation 
requirements
    for n = size(xr,2):-1:1
        xr(:,n,m) = decimate(dechirpsig(:,n),Dn,'FIR');
    end
end

%VIRTUAL ARRAY PROCESSING

%data cube received by the physical RX array but be 
processed to form the
%virtual array cube. Since the measurements taken 
correspond to 2 TX
%antennas, elements can be recovered from 2 consecutive 
sweeps

Nvsweep = Nsweep/2;
xr1 = xr(:,:,1:2:end); %returns corresponsing to the 
1st TX element
xr2 = xr(:,:,2:2:end); %returns corresponding to 2nd TX 
element

%create virtual array

xrv = cat(2, xr1,xr2);

%remove the third dimension of xrv since it's not 
required for this simulation
xrv = xrv(:,:,1);
```

```
%%CPU MVDR COMPUTATION

%create sterring vector to hold phase differences
between elements in the
%virtual array
%
steervec =
phased.SteeringVector('SensorArray',virtual_array);
sv = steervec(fc,teta);

Rxx = transpose(xrv)*xrv;

%carry out MVDR Algorithm
tic %start timer
S=diag(transpose(sv)*(Rxx\sv));
toc %stop timer

S_dB=-10*log10(S);

%PLOT SPECTRUM
figure;
plot(teta,S_dB);
title('CPU Version- MVDR Spectrum');
xlabel('Angle(degrees)');
ylabel('Power (dB)');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%

%%GPU MVDR CALCULATION

%copy necessary data to the GPU
sv_parallel = gpuArray(sv);
Rxx_parallel = gpuArray(Rxx);

%carry out MVDR algorithm
tic %start timer
S_parallel=diag(transpose(sv_parallel)*(Rxx_parallel\sv
_parallel));
%S_parallel=diag(S_parallel);
toc %stop timer
```

```
%TRANSFER DATA BACK TO MATLAB WORKSPACE
mvdr_spectrum_gpu = gather(S_parallel);
mvdr_spectrum_gpu_dB=-
10*log10(abs((mvdr_spectrum_gpu)));

%PLOT SPECTRUM

figure;
plot(teta,mvdr_spectrum_gpu_dB);
title ('GPU Version- MVDR Spectrum');
xlabel('Angle(degrees)');
ylabel('Power (dB)');

function wav = helperDesignFMCWWaveform(c,lambda)
% This function is provided by Mathworks as part of
their MIMO Radar
% Virtual Array Example file to simulate a FMCW
waveform. It was used in
% this simulation to create the same wave.

% Copyright 2017 The MathWorks, Inc.

range_max = 200;
tm = 5.5*range2time(range_max,c);
range_res = 1;
bw = range2bw(range_res,c);
sweep_slope = bw/tm;
fr_max = range2beat(range_max,sweep_slope,c);

v_max = 230*1000/3600;
fd_max = speed2dop(2*v_max,lambda);

fb_max = fr_max+fd_max;
fs = max(2*fb_max,bw);
wav =
phased.FMCWWaveform('SweepTime',tm,'SweepBandwidth',bw,
...
    'SampleRate',fs);

end
```