GPU ACCELERATED RADIO WAVE PROPAGATION MODELING USING RAY TRACING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ALAETTİN ZUBAROĞLU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2014

Approval of the thesis:

GPU ACCELERATED RADIO WAVE PROPAGATION MODELING USING RAY TRACING

submitted by **ALAETTİN ZUBAROĞLU** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Name, Last Name:	ALAETTİN ZUBAROĞLU
Signature :	

ABSTRACT

GPU ACCELERATED RADIO WAVE PROPAGATION MODELING USING RAY TRACING

Zubaroğlu, Alaettin
M.S., Department of Computer Engineering
Supervisor : Dr. Cevat Şener
Co-Supervisor : Assoc. Prof. Dr. Tolga Can

September 2014, 71 pages

Radar producers, which are mostly in defense industry, need radar environment simulator to test their products during the development. Such a simulator helps them to be able to get rid of costly field tests. For developing a radar environment simulator, radio wave propagation should be modeled. However, this is a computationally expensive and time consuming process. Improving the performance of propagation modeling contributes to the radar development work.

Ray tracing is one of the several electromagnetic wave propagation techniques. It enables calculation of total range, delay and power of radio waves on each point of the field. In this study, we have developed a radio wave propagation modeling application using a parallel implementation of the ray tracing method. Reflection, refraction and free space path loss properties of radio waves are implemented. Predefined atmosphere types that affect the refraction and surface types that affect the reflection are included for user selection. Moreover, the user has the chance of defining special atmosphere and surface types. Our application works on two-dimensional (2D) maps. It also has the ability of converting three-dimensional (3D) maps to 2D slices and working on them.

We have developed and accelerated the application using GPU computing and parallel programming concepts. We have run the proposed method sequential and parallel on

CPU and parallel on GPU. We have compared and analyzed time measurements of the application on different domains. We have achieved up to 18.14 speedup values between high specification CPU and GPU cards within the scope of this thesis.

Keywords: Ray Tracing, GPU, CUDA, Radar, Radar Wave, Wave Propagation, Propagation Modeling, Radar Environment Simulator

ÖZ

GPU HIZLANDIRMALI IŞIN İZLEME İLE RADAR DALGA YAYILIMI MODELLEMESİ

Zubaroğlu, Alaettin
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Dr. Cevat Şener
Ortak Tez Yöneticisi : Doç. Dr. Tolga Can

Eylül 2014 , 71 sayfa

Çoğunlukla savunma sanayi alanında faaliyet gösteren radar üreticileri, geliştirmekte oldukları ürünleri denemek için radar ortam simülatörüne ihtiyaç duyarlar. Bu tarz bir simülatör, üreticilerin pahalı saha testlerinden kurtulmalarına yardımcı olur. Radar ortam simülatörü geliştirmek için, radyo dalgalarının yayılımının modellenmesi gerekmektedir. Yalnız bu, bilgisayarlar için hesaplanması zor ve zaman alıcı bir işlemdir. Yayılım modellemesi işleminin hızlandırılması radar üretimine katkı sağlar.

Işın izleme, elektromanyetik dalga yayılımı modelleme yöntemlerinden biridir. Dalganın, arazinin her noktasındaki toplam katettiği yolu, gecikmesini ve gücünü hesaplamaya olanak sağlar. Bu çalışmamızda, ışın izleme yöntemi ile radyo dalgaları yayılımını modelleyen bir uygulama geliştirdik. Yansıma, kırılma ve boş alan kaybı hesapları bu proje kapsamında geliştirilmiştir. Bununla birlikte, uygulamamızda kırılmayı etkileyen öntanımlı atmosfer tipleri ve yansımayı etkileyen öntanımlı yüzey tipleri kullanıcı seçimleri için bulunmaktadır. Ayrıca kullanıcıya yeni bir atmosfer tipi ve zemin tipi tanımlama esnekliği sağlanmıştır. Uygulamamız 2 boyutlu haritalarda çalışmak üzere geliştirilmiştir. Ancak 3 boyutlu haritalardan 2 boyutlu dilimler hesaplayarak bu 2 boyutlu dilimler üzerinde çalışma yeteneğine de sahiptir.

Uygulamamızı ekran kartı programlama yöntemleri ile paralel biçimde hızlandırdık. Geliştirmiş olduğumuz uygulamayı işlemci üzerinde seri, işlemci üzerinde paralel ve ekran kartı üzerinde paralel olarak çalıştırdık. Uygulamanın farklı ortamlardaki çalışma sürelerini karşılaştırıp analiz ettik. Bu çalışma kapsamında, yüksek teknik özellikli işlemci ve ekran kartları arasında 18.14 kata kadar hızlanma yakaladık.

Anahtar Kelimeler: Işın İzleme, Ekran Kartı, CUDA, Radar, Radar Dalgaları, Dalga Yayılımı, Yayılım Modelleme, Radar Ortam Simulatörü

To each of my family members and my darling future wife

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Dr. Cevat Şener and co-supervisor Assoc. Prof. Tolga Can for their guidance and contributions at all stages of this work. It is a great honor to work with them.

I am grateful to Osman Günay for his contributions and patience in answering my never ending questions.

I am indebted to my parents, Suphi Zubaroğlu and Azize Zubaroğlu, for their continuous support, confidence in me, encouragement and love throughout my life. You have both made substantial sacrifices to help me attain my goals and you will never know the degree of my appreciation or admiration of you.

I am grateful to my siblings, Güler İstanbullu, Tahsin İstanbullu, Arif Zubaroğlu, Nesrin Zubaroğlu and Tahsin Zubaroğlu for their continuous support and encouragement throughout my life.

It is a pleasure to express my special thanks to my love, Pelin, for her endless love and understanding.

And lastly, I would like to thank İris İstanbullu, Sara Ada İstanbullu and Mira Zubaroğlu for just being my nieces.

The work presented in this thesis is co-financed by Republic Of Turkey, Ministry of Science, Industry and Technology (75%) and Microwave Electronic Systems, Inc.-MIKES (25%) within the scope of SANTEZ Programme with the project number 01338.STZ.2012-1.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

kHz	Kilo Hertz
MHz	Mega Hertz
GHz	Giga Hertz
FSPL	Free Space Path Loss
NFS	Network File System
GPU	Graphics Processing Unit
MPI	Message Passing Interface
CPU	Central Processing Unit
SMX	Streaming Multiprocessor
L1	Level 1
L2	Level 2
ALU	Arithmetic Logic Unit
FPU	Floating Point Unit
CUDA	Compute Unified Device Architecture
KB	Kilo Byte
LD/ST	Load/Store Unit
SFU	Special Function Unit
DP	Double Precision
RWPRay	Radio Wave Propagation Using Ray Tracing
2D	Two Dimensional
3D	Three Dimensional
RS	Ray Set

CHAPTER 1

INTRODUCTION

1.1 Problem Definition

Radars are object detection systems. They use radio waves to determine some properties of objects, like range, altitude, speed and direction. Radars can detect aircrafts, ships, missiles, vehicles, terrain or other similar objects.

Radio Waves, are a type of electromagnetic radiation that have frequencies from 3 kHz to 300 GHz [12]. They travel at the speed of light like all other electromagnetic waves. Radio waves have propagation characteristics that vary by frequencies. However, for all frequencies, radio waves are being affected from the Earth's atmosphere, and the objects they encounter while traveling.

To be able to simulate the radar environment (the Earth with the atmosphere, terrain and external objects), radio wave propagation must be modeled. One of the commonly used ways of modeling radio wave propagation is ray tracing [6]. Ray tracing is a computationally expensive and time consuming method. Accelerating implementations of this method are welcomed by the radar researchers. Speeding up radio wave propagation modeling means both similar accuracy results in comparison to sequential models in less duration and better, more accurate results in the same duration.

Radar environment simulation is important for radar system developers. Such a simulator helps the developers to test and implement their product faster and easier with no need for costly field tests. It also gives developers the opportunity to test the system with exactly same parameters and environmental conditions which is not actually possible in the real world. Accelerating radar environment simulation will result in speeding up radar system developments, and this is possible with faster radio wave propagation modeling.

1.2 Motivation

Radar systems are important for national defense. Local defense industry companies are already working on radar systems and surely they will keep working. While technology and warfare systems improve everyday, each country needs more reliable and up-to-date radar systems with all other electronic warfare systems.

While developing and improving radar systems, developers encounter different difficulties, some of which are about testing. To test the radar systems, developers must take the actual system to the field and set up all the system there. Beside of all difficulties of working on the field, debugging is even less possible, because of the test unrepeatability. Radar system tests are directly dependent on environmental and climatic conditions. Each run of any tests will become a new test because of the changing conditions in the real world. It is not possible to do the same test with exactly same parameters twice. This situation results in long debugging time and still unrealized bugs.

A radar environmental simulator system, can be developers' best friend and helper in the radar system development and testing process. Using such a system, developers can test and debug their radar systems in a more comfortable working place, which is their own office or laboratory and with the opportunity of test repeatability. Doing the same test with exactly same parameters and environmental conditions is possible with such simulator systems. Such a tool would accelerate development process of radar systems.

The most complex and time consuming part of radar environment simulation is the radio wave propagation modeling part. Hence, speeding up this part will directly speed up the whole simulator system. As might be expected, faster simulator systems help the radar developers more and they are accepted instead of slower ones by the developers.

As a result, using radar environmental simulator systems saves time, labor and budget of radar developers.

1.3 Related Work

There are several studies about ray tracing for electromagnetic wave propagation modeling, some of which are [4, 5, 2, 17]. These four studies are described briefly in this section.

A parallel ray tracing model for radio wave propagation prediction in 3D is described in [4]. In Cavalcante et al.'s study, shooting and bouncing ray method is used. The 3D source modeling strategy that is used is adopted from [16]. In that method, the ray creation at the transmitter is based on a technique where a regular icosahedron is inscribed in a unit sphere surrounding the radar. Reflection (described in Section 2.1.1) and diffraction (described in Section 2.1.3) properties of the waves are implemented within the scope of Cavalcante et al.'s project. For parallelization, equal number of initial rays are distributed to different nodes of a cluster and this distribution is done using network file system (NFS). A flat terrain is assumed , and propagation of the waves in a small area among the buildings is modeled.

In [5], a study to accelerate ray tracing for electromagnetic propagation analysis on graphics processing unit (GPU) is presented. Shooting and bouncing ray method is used in Epstein et al.'s study too. Reflection and diffraction are implemented. Initial power of the rays are obtained from antenna pattern data. Ray power and number of reflections are the two termination criteria that are used. For distribution of the job, separate GPU cores are assigned to separate azimuth angles, and each of the cores handles all of the elevation angles. There is no terrain information and the scene that is used is composed of geometrical elements which are the buildings.

Another method about usage of parallel ray tracing for radio wave propagation modeling is proposed in [2]. The main concept of Athanaileas et al.'s study is the creation of a tree of images of the transmitter point. This process is parallelized in that study with the concern of scalability, efficiency, task granularity, reduction of communication, task distribution and load balancing. Parallelization of it is based on message passing interface (MPI). Dynamic load balancing is provided by means of the master-worker and the work-pool patterns. Reflection, diffraction and foliage attenuation are implemented within the scope of Athanaileas et al.'s project.

Lastly, an application of ray tracing for electromagnetic wave propagation simulation is described in [17]. It is a ray shooting Matlab package that works in 2D. Propagation (described in Section 2.1), reflection and refraction (described in Section 2.1.2) are implemented within the scope of Sevgi's study. It differs from [4], [5] and [2] because refraction is not implemented in them. In that application, reflection occurs from the bottom surfaces and from the top of the obstacles but not from the sides of the obstacles. The ray that hits left part of any obstacles ends there. Backward propagation is not implemented. For the refraction property, it is possible to define the atmosphere type and make the application calculate the refractive indices of the region. Sevgi's application is the most similar one to our study and we have used it for verification of our results.

In our thesis, in addition to Sevgi's work, we have implemented backward propagation, surface dependent reflection and terrain height information. Backward propagation enables the ray to reflect from upright terrain and propagate in reverse direction. We have defined different surface types and used reflection properties of them as described in Section 3.3.2. Sevgi's application works among basic shapes which simulate the buildings, however our study works on terrain height maps. Moreover, we have also developed a technique to run our study on 3D maps, which is described in Section 3.1. Lastly, the main objective of our thesis is to accelerate the radio wave propagation modeling. We use GPU for that purpose.

1.4 Contributions

There are two main contributions in this study.

The first contribution is that we have implemented a radio wave propagation modeling tool for CPU and GPU, and we achieved up to 18.14 speedup values on GPU. In this study, we have accelerated the radar wave propagation simulation 18.14 times.

The second contribution is that we have implemented a new method for parallel 3D ray tracing. We have created 2D slices, worked on them, and combined the results. 2D slicing is a new technique for 3D ray tracing.

1.5 Outline of the Thesis

The rest of the thesis is organized as follows: Chapter 2 provides background information about radio wave propagation, ray tracing and graphics processing unit (GPU). Chapter 3 details our radio wave propagation modeling application, RWPRay. Chapter 4 gives the implementation details of RWPRay on CPU and GPU. Chapter 5 provides test environment and data sets used in experiments. It also presents experimental results and analysis. Chapter 6 concludes the thesis, provides a summary of key findings, and gives suggestions for future research.

CHAPTER 2

BACKGROUND

2.1 Radio Wave Propagation

Radio wave propagation, is the radio waves' behavior of transmission from one point on the Earth, which is called transmitter, to the other, that is called receiver.

Radio waves, like any other electromagnetic waves, are being affected from the climatic conditions and any objects in the area, while they are propagating. Terrain, mountains, foliage, humidity, cloudiness, wind and temperature are the first natural factors coming to mind that affect the propagation. In addition, buildings, vehicles and all other external objects, are human made propagation affecting factors.

Environmental effects on the radio wave propagation are diverse, however they can generally attributed to *reflection, refraction, diffraction, absorption* and *scattering* [15]. Electromagnetic waves can be described as vectorial quantities. In other words, they have both direction and power. Environmental effects may, and mostly do, change both direction and power of radio waves. Radio waves are able to reach many points and fields that are not in the line-of-sight area under favor of these effects [3].

Radio waves, lose their power while propagating. Power of the wave is calculated according to free space path loss formula [20, 21]. Free space path loss is the loss in power of the wave that will result from a path through free space (air), with no objects nearby to cause reflection or diffraction. This formula is only accurate in the far field where spherical spreading can be assumed, it is not correct for the field close to the transmitter[19]. The formula is as follows:

$$fspl = \left(\frac{4\pi df}{c}\right)^2 \tag{2.1}$$

where:

$fspl$ **:** Free-space path loss

f **:** Signal frequency (in hertz)

d **:** Distance from the transmitter (in meters)

c **:** Speed of light in vacuum (in m/s, 3e8 m/s)

2.1.1 Reflection

When a radio wave hits a smooth surface while propagation, it partially reflects and partially transmits. For the reflected wave a regular reflection occurs on that surface. The ray reflects from the surface with the same angle. Smooth surfaces act like mirrors for electromagnetic waves and this creates a regular reflection. Radio waves lose power while reflecting from smooth surfaces. Power of reflected wave is surely less than the incident wave's. New power of the reflecting wave depends on reflection coefficient of the surface. Reflection coefficient is the ratio of the power of the reflected wave to the power of the incident wave. Hence, to calculate power of the reflected wave, power of the incident wave is directly multiplied by the reflection coefficient. In addition, transmission coefficient of the surface is the ratio of the power of the transmitted wave to the power of the incident wave. However, transmitted partial of the wave is generally neglected. Reflection coefficient and transmission coefficient are closely related to each other. Figure 2.1 describes the reflection. More information can be obtained from [1, 3, 20, 21].

2.1.2 Refraction

Each medium has its refraction index and this is another important characteristic factor of the area for radio wave propagation. When a radio wave moves into a medium that has a different refraction index from the source medium, again the wave partially

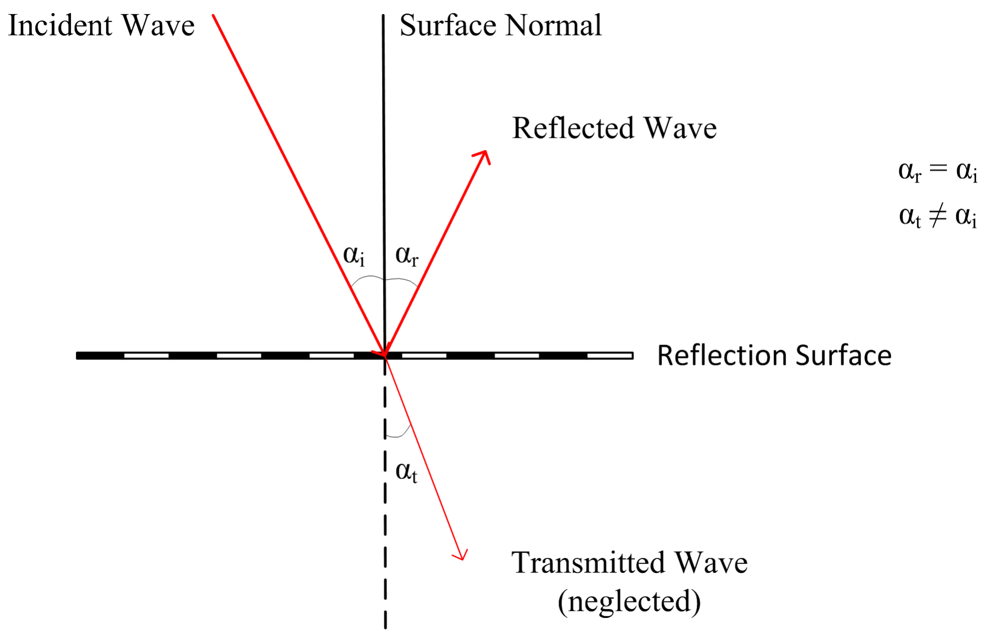


Figure 2.1: Reflection

reflects and partially transmits. When none of the mediums is a solid, opaque object, the reflecting partial is powerless and mostly neglected. The transmitted partial is the more powerful and continuing one. However, because direction of the wave changes, it is called to be refracted rather than transmitted. While the reflected partial is neglected, the refracted partial is accepted to have the same power with the incident wave. The wave does not lose power because of refraction, however its power reduces depending to the length of the path it travels according to free space path loss formula. Figure 2.2 describes the refraction.

Angle of the refracted wave is calculated according to Snell's Law [17, 3, 21]. When a radio wave strikes the interface between two mediums which have refraction indices η_1 and η_2 respectively, angle of refracted wave is calculated as follows:

$$\frac{\sin \alpha_i}{\sin \alpha_t} = \frac{\eta_2}{\eta_1} \tag{2.2}$$

2.1.3 Diffraction

When radio waves meet an obstacle like buildings, hills or any other solid, opaque objects, they are blocked and in the same time diffracted. With the irregularities of

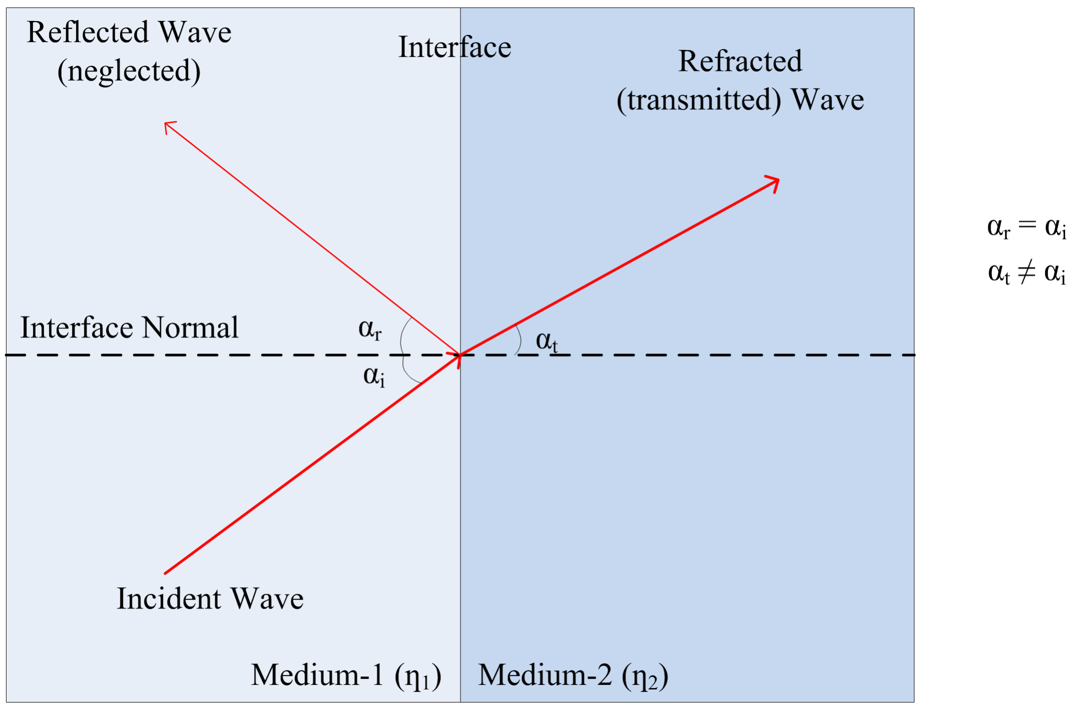
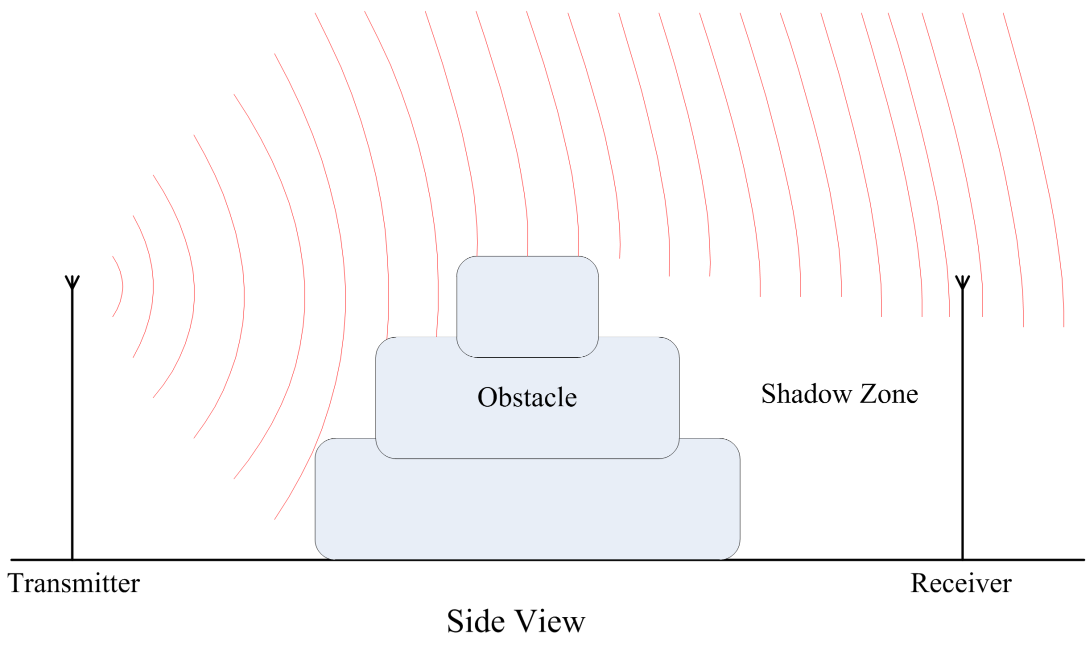


Figure 2.2: Refraction

diffraction, unblocked waves create secondary waves in some fields even behind the obstacles. As a result of diffraction, radio waves may be received by receivers that are not in line of sight field. Despite the diffraction, still there may be fields with no received radio wave, which is called *shadow region*. Figure 2.3 describes the diffraction.

2.1.4 Scattering

Scattering is the process that the radio wave is forced to deviate from its path by more than one paths because of non uniformities on the surface that the radio wave reflects. Scattering differs from the reflection due to the roughness of the surface. On rough surfaces, while reflecting the radio wave splits into partial waves and loses the grater part of its power depending on the roughness. The more nonuniformities the surface have, the more the radio wave splits into partial waves and lose their power. When the surface is slightly rough, the process is called *coherent scattering* and when the surface is very rough, it is called *diffuse scattering* [22]. Figure 2.4 describes the scattering.



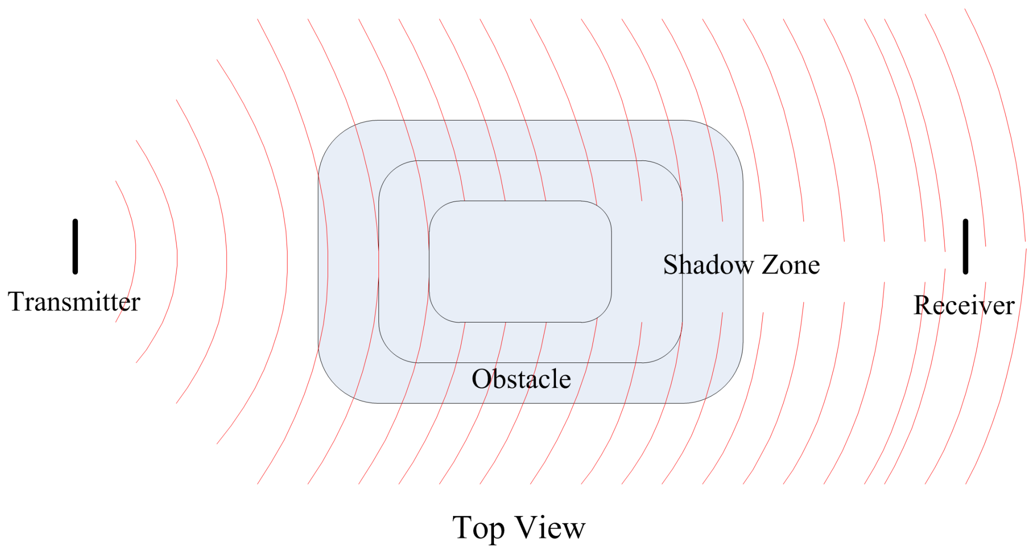


Figure 2.3: Diffraction

2.1.5 Absorption

Absorption of a radio wave is the loss in the power of the wave while it is traveling in a medium or reflecting from a surface. The reason of the absorption is that, the matters in the field get little part of the energy from the wave. Absorption is also called

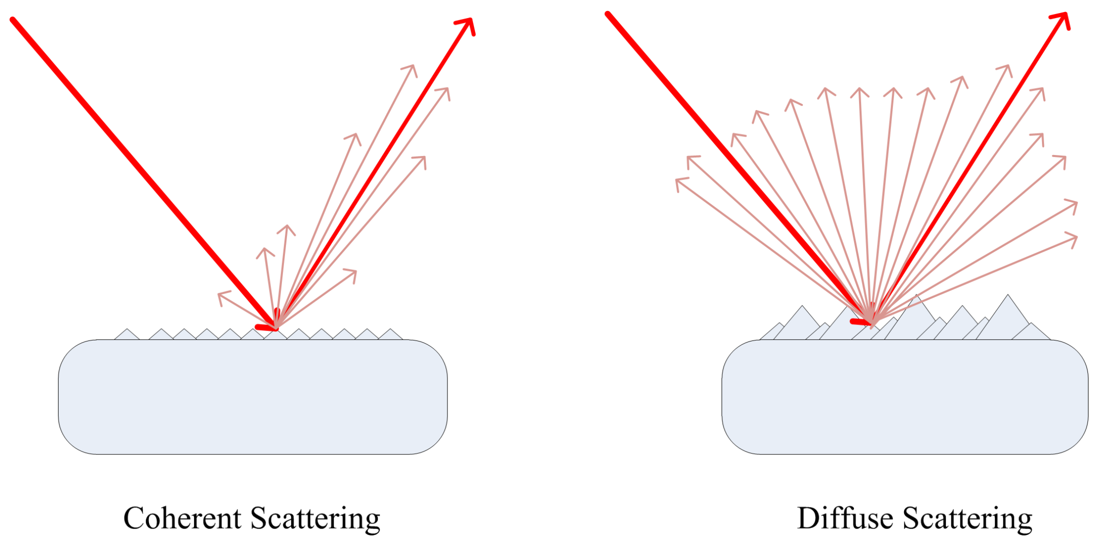


Figure 2.4: Scattering

attenuation. Absorption during reflection is calculated by the reflection coefficient, and that is during propagation is calculated by the free space path loss equation.

2.2 Ray Tracing

Ray tracing, is a technique to calculate the path of waves or particles through a system. This technique is practiced in two distinct forms which are ray tracing in computer graphics and ray tracing in physics.

2.2.1 Ray Tracing in Graphics

In computer graphics, ray tracing is a method that is being used to generate the image that will appear on the screen [18]. In this method, the paths from the eye (camera) to each pixel are being traced, and the color of that pixel is determined according to the object these paths encounter. One different ray for each pixel is created and sent. These rays continue to determine the color of that pixel according to the positions of the objects and light sources. This method is capable of producing high degree of visual realism which is higher than other rendering techniques. However, ray tracing is computationally more expensive than other techniques. Because of its high complexity, ray tracing is more suitable for applications that is possible to render the image slowly ahead of time. It is poorly suited to real time applications because of the need for high speed [23]. Figure 2.5 describes ray tracing for image rendering.

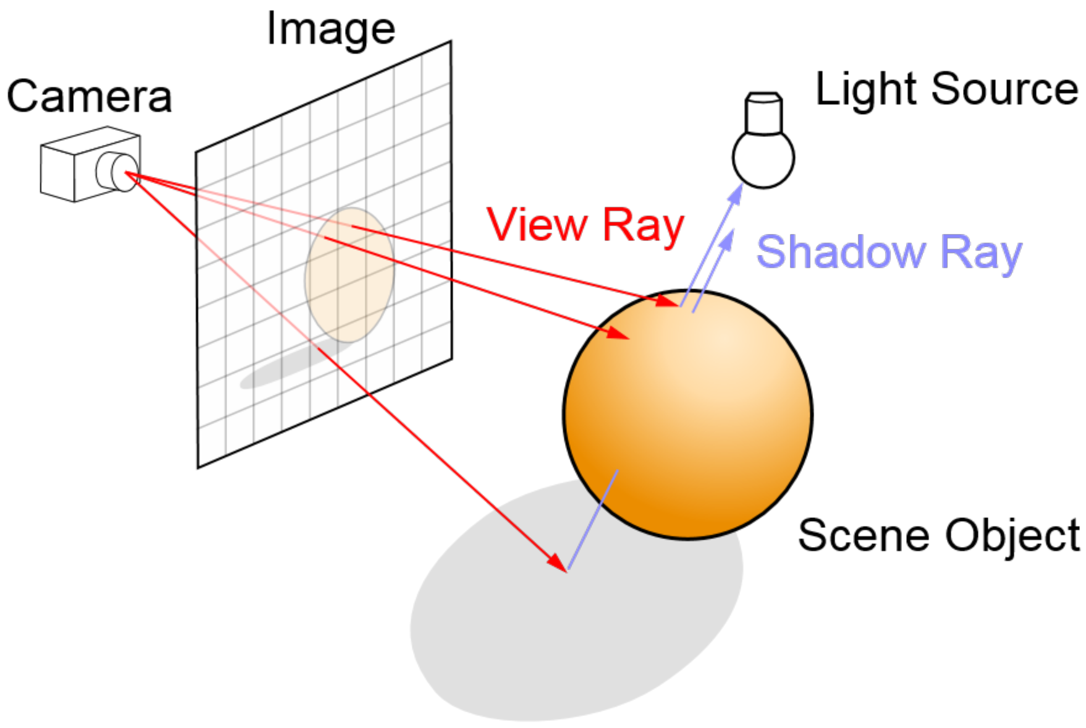


Figure 2.5: Ray tracing algorithm for rendering an image. Adapted from K, Henri. "Ray Tracing." Wikipedia. Wikimedia Foundation, 12 Apr. 2008. Web. 23 July 2014

2.2.2 Ray Tracing in Physics

Ray tracing in physics is a technique that is being used to calculate the traveling path of electromagnetic waves in an area that have regions with different propagation velocities, obstacles and reflective objects [9]. The propagation velocity defines refraction index of a region. Effect of regions with different propagation velocities is change of direction which is also called refraction, effect of obstacles is dying out and effect of reflective objects is reflection, which has results on both the direction and the power of the wave.

This ray tracing technique, models the wave as a large number of very narrow beams. Rays correspond to these beams. Rays travel straight until the refractive index, which is dependent to the propagation velocity, changes. When such a change exists, new direction of the ray is calculated according to Snell's Law [17]. While traveling, rays are checked whether any collisions occur with the solid objects in environment. In case of collisions, a regular reflection takes place. According to regular reflection, reflection angle is equal to incoming angle.

Rays lose power while traveling. Power of the ray on a selected position is dependent to the distance from the transmission point. This situation is also called attenuation. Power loss of each step is calculated by free space path loss equation using the step range. Besides, rays lose power during the reflection too. Each reflective object has its own reflection coefficient and power of the ray is multiplied by that characteristic property when the ray hits that object.

Ray tracing is a computationally expensive and time consuming technique because of its high complexity level. However, it is embarrassingly parallel. This means, ray tracing is simple to parallelize because of the independence between its small and identical parts. Each ray is calculated independently from all others and result of each ray is also stored by itself. There is no communication requirement between the rays during the calculation or after [24, 17, 21]. Figure 2.6 describes ray tracing in physics.

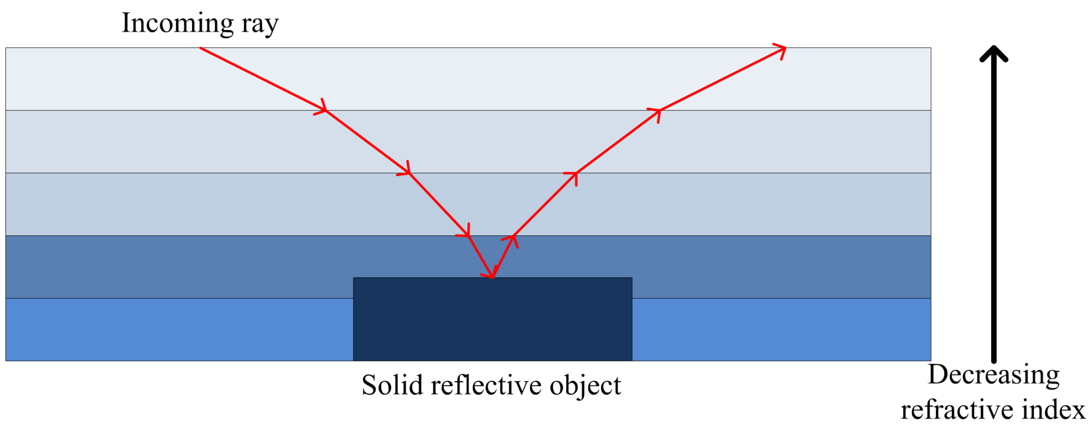


Figure 2.6: Ray Tracing of a beam passing through a medium with changing refractive index

2.3 Graphics Processing Unit

A Graphics Processing Unit (GPU) is an electronic circuit that is specially designed to accelerate the image creation in sequence, for displaying them on a monitor. Modern

GPUs are very efficient at manipulating computer graphics and they have a highly parallel structure. This structure makes them more effective than ordinary general purpose Central Processing Units (CPU) for special solutions. We can define these special solutions as algorithms which are processing a huge amount of data in parallel. The first GPU in the world, GeForce 256, is produced and marketed by Nvidia in 1999.

2.3.1 GPU Computing

GPU computing is the use of graphics processing units in general purposes. It is being used together with a CPU to accelerate engineering, scientific or any other high performance needing applications. General purpose GPU programming is started in 2007 and it still powers various data and computation centers. GPUs are specialized to do hundreds of small, identical jobs in parallel. Data parallel problems and algorithms are more adoptable to GPU programming because they apply the same instructions to different data parts and this is exactly what GPU can do best.

One of the challenges in general purpose GPU programming is the memory optimization. Mostly there exists less amount of memory and less cache space on GPU cards than the whole system. The programmer must fit into the memory space, it is not possible to exceed it. Moreover, the user should also optimize the memory usage to get high performance.

2.3.2 Architecture

Graphics processing unit has a special architecture that is designed for high performance visualization processing. It has hundreds, even thousands of cores working all together. It has a dedicated memory which is called global memory and this is accessible by all the cores. L2 cache, which is a faster global memory, may also exist on the card, and it is accessible by all cores. Communication with the main system (CPU) is done using the global memory. Cores together constitute multiprocessors, which are newly began to be called streaming multiprocessors and abbreviated as SMX. Each streaming multiprocessor may contain cores, a shared memory called L1 cache, load-

/store units, special function units which are designed for double precision and wrap schedulers. L1 cache is shared among the processors belong to that multiprocessor. Moreover, each core has its own arithmetic logic unit (ALU) and floating point unit (FPU). Details of GPU architecture varies from card to card. However, global memory, shared memory, multiprocessor and core concepts are similar for most of them.

Figure 2.7 shows Nvidia Fermi Architecture [10]. This card has 16 streaming multiprocessors and architecture of them is being showed in Figure 2.8. Each fermi multiprocessor has 32 CUDA cores, 16 load/store units, 4 special function units, dual warp schedulers, 64 KB configurable shared memory and L1 cache.

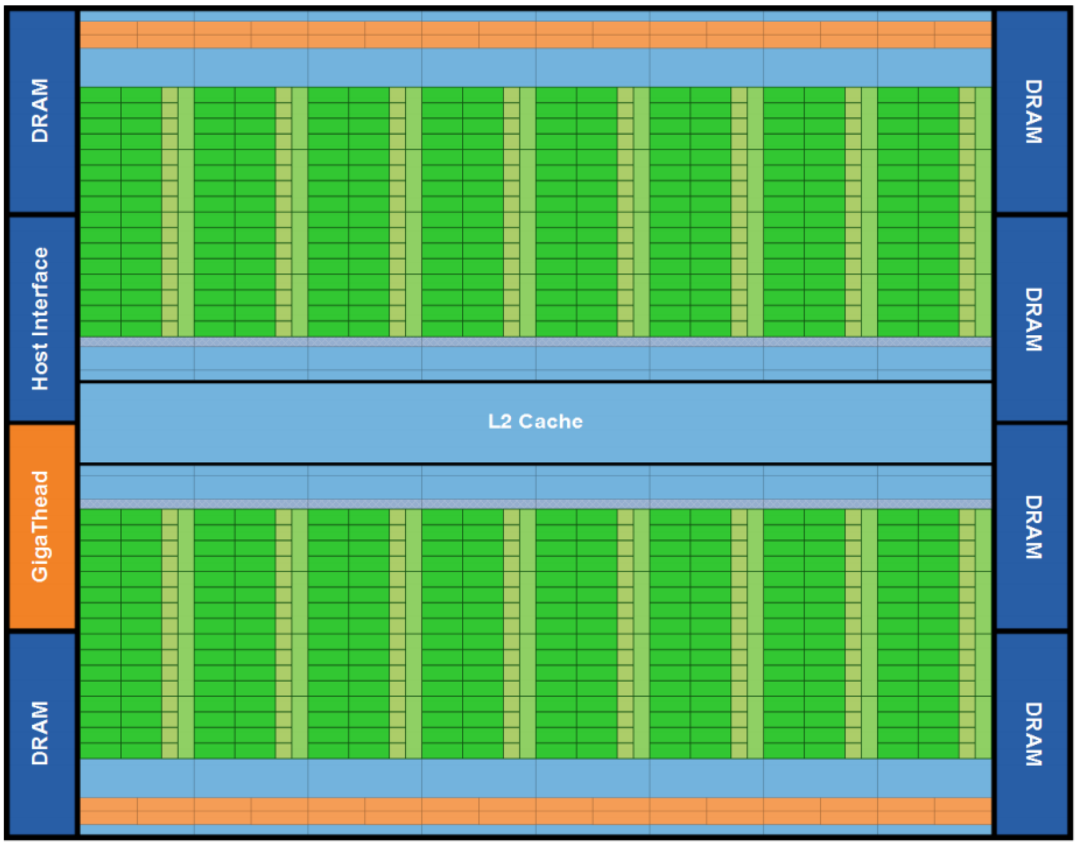


Figure 2.7: Nvidia Fermi Architecture. Adapted from "NVIDIA Fermi Compute Architecture Whitepaper", by Nvidia Corporation, 2009, p. 7. Copyright 2009 by Nvidia Corporation [10]

Figure 2.9 shows Nvidia Kepler Architecture [11]. Kepler GK110 card has 15 streaming multiprocessors and architecture of them is being showed in Figure 2.10. Each kepler multiprocessor has 192 single precision CUDA cores, 64 double precision units,

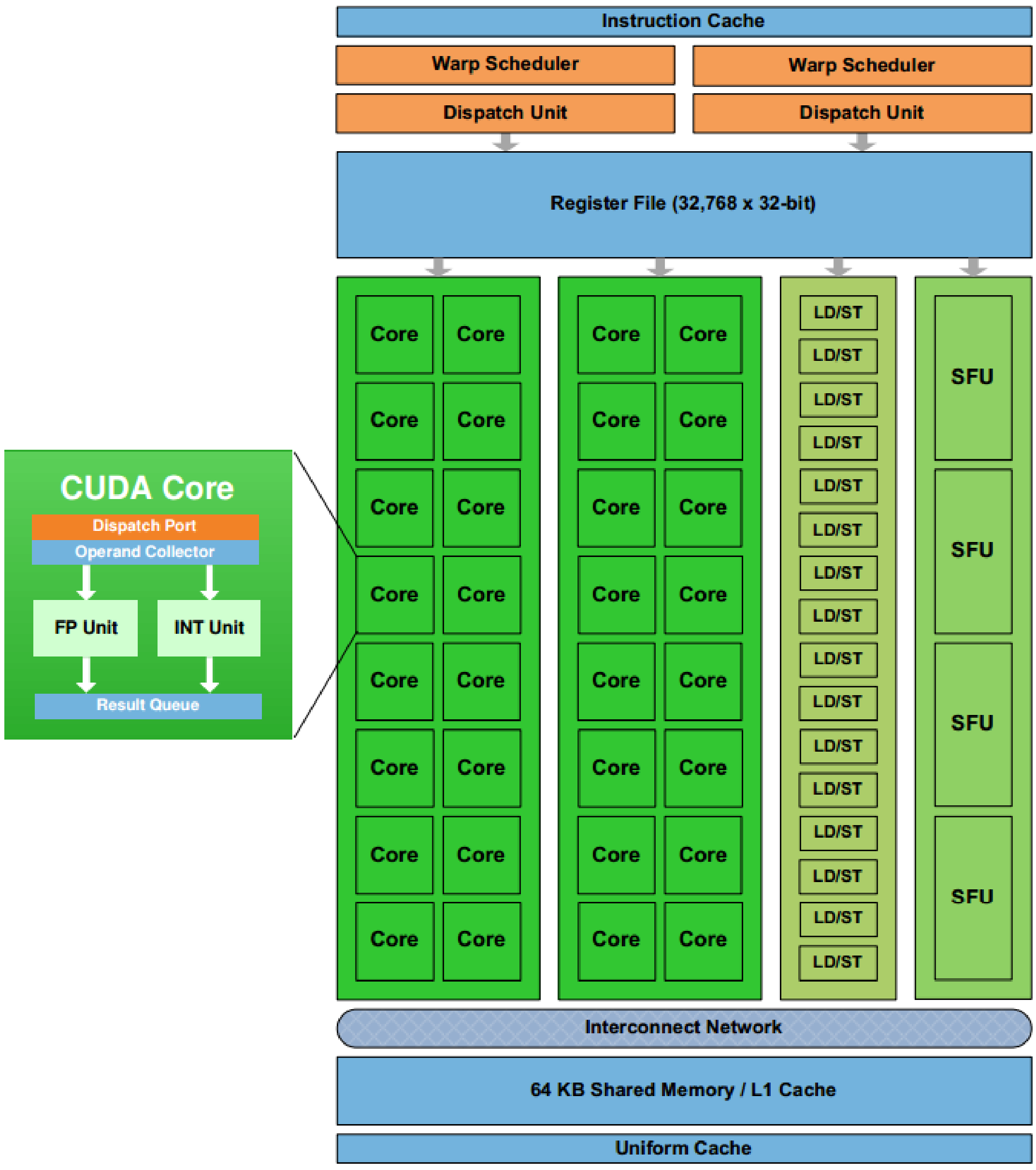


Figure 2.8: Nvidia Fermi Streaming Multiprocessor Architecture. Adapted from "NVIDIA Fermi Compute Architecture Whitepaper", by Nvidia Corporation, 2009, p. 8. Copyright 2009 by Nvidia Corporation [10]

32 load/store units, 32 special function units, quad warp schedulers, 64 KB configurable shared memory and L1 cache and 48 KB read only data cache.

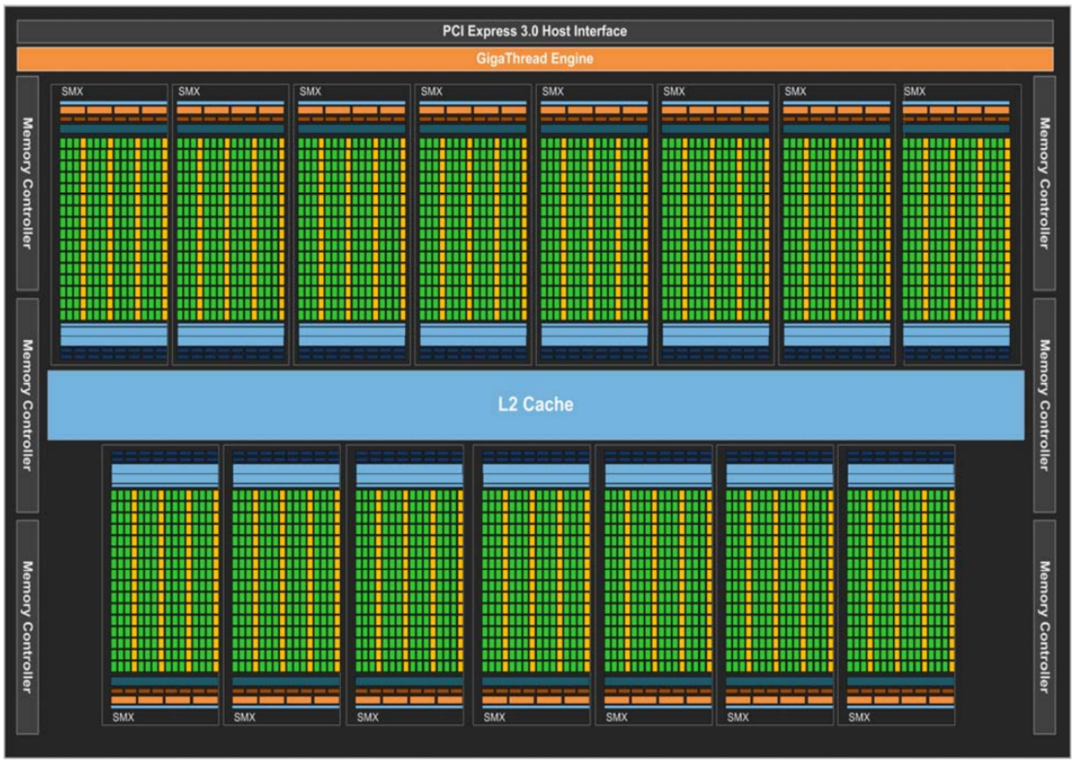


Figure 2.9: Nvidia Kepler Architecture. Adapted from "NVIDIA Kepler GK110 Architecture Whitepaper", by Nvidia Corporation, 2012, p. 6. Copyright 2012 by Nvidia Corporation [11]

2.3.3 CUDA

Compute Unified Device Architecture (CUDA) is the general purpose GPU programming model invented by Nvidia. It enables the user to use the GPU for engineering, scientific or any other general purpose applications in parallel computing manner. It gives the programmer the chance of sending a parallelized job to the GPU to take advantage of GPU's manycore architecture. CUDA can be used with C, C++ and Fortran languages.

Source code or flow of a GPU computing application does not differ from a standard CPU application very much. GPU computing application has an extra kernel function which runs parallel on the GPU. The application starts on the CPU, reads user parameters or files if any exist, does the initializations and runs the sequential part of the application there. When parallel working is needed, the application transfers the required data to GPU memory and starts the kernel function. When work of the

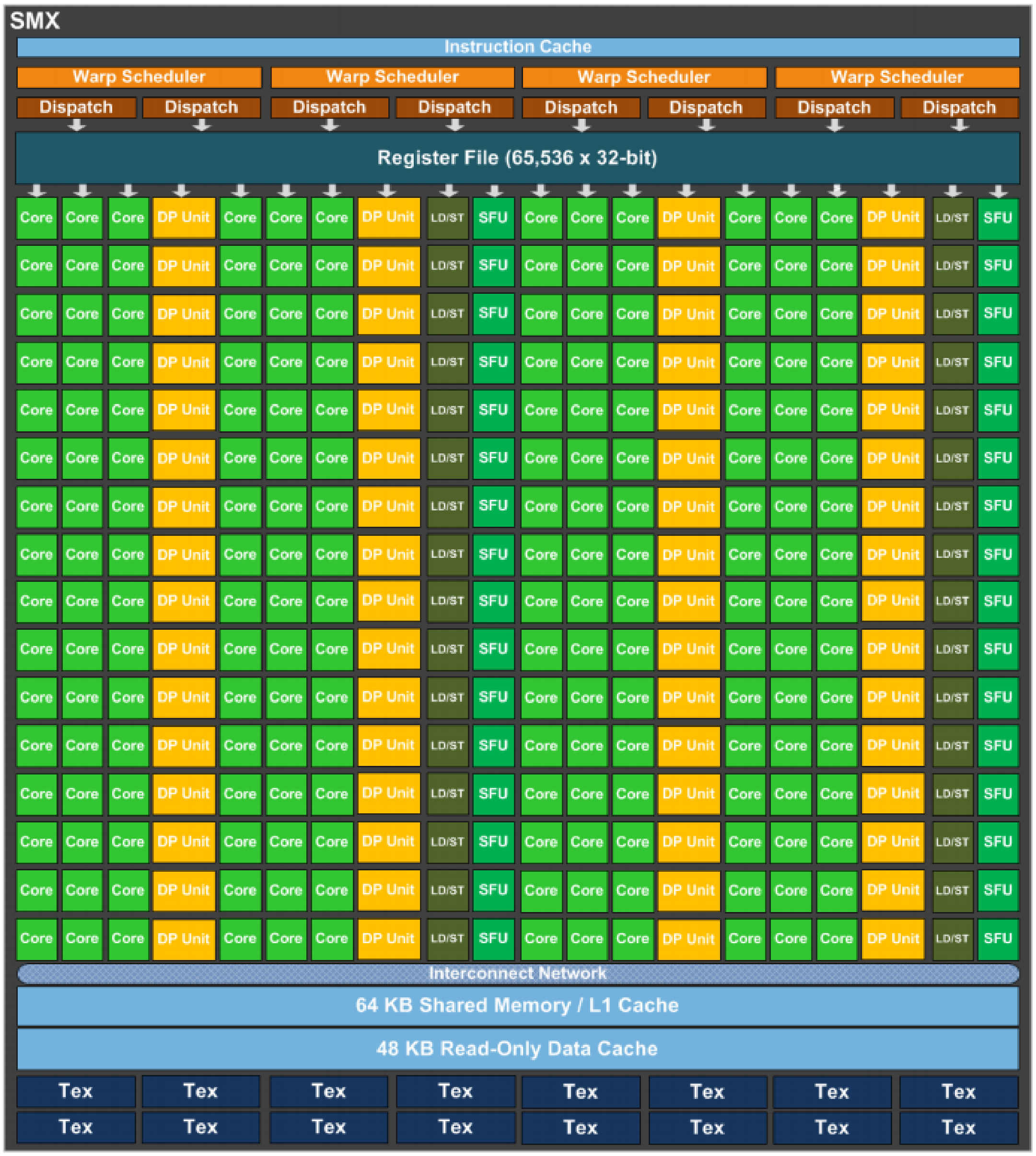

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Figure 2.10: Nvidia Kepler Streaming Multiprocessor Architecture. Adapted from "NVIDIA Kepler GK110 Architecture Whitepaper", by Nvidia Corporation, 2012, p. 8. Copyright 2012 by Nvidia Corporation [11]

GPU is finished, application transfers the resultant data from GPU memory to CPU memory. Rest of the application is a standard CPU program.

GPU is a manycore system. It hosts hundreds of cores and the same copy of the kernel function runs on these cores. Each copy is called a CUDA thread. Group of CUDA threads is called a block and group of blocks is called a grid. To run the CUDA kernel, programmer defines grid size and block size parameters. Grid size is the block count, and block size is the thread count in a block.

Threads have in block unique IDs and blocks have in grid unique IDs. Unique IDs of the threads in the whole system can be calculated using the block and thread IDs. Each thread has its own private local memory and each block has its own per block shared, and per grid private memory. The threads within the same block can communicate using this shared memory part. Global memory of the GPU is accessible by all threads. Figure 2.11 describes CUDA hierarchy of thread, blocks and grids with corresponding memory spaces.

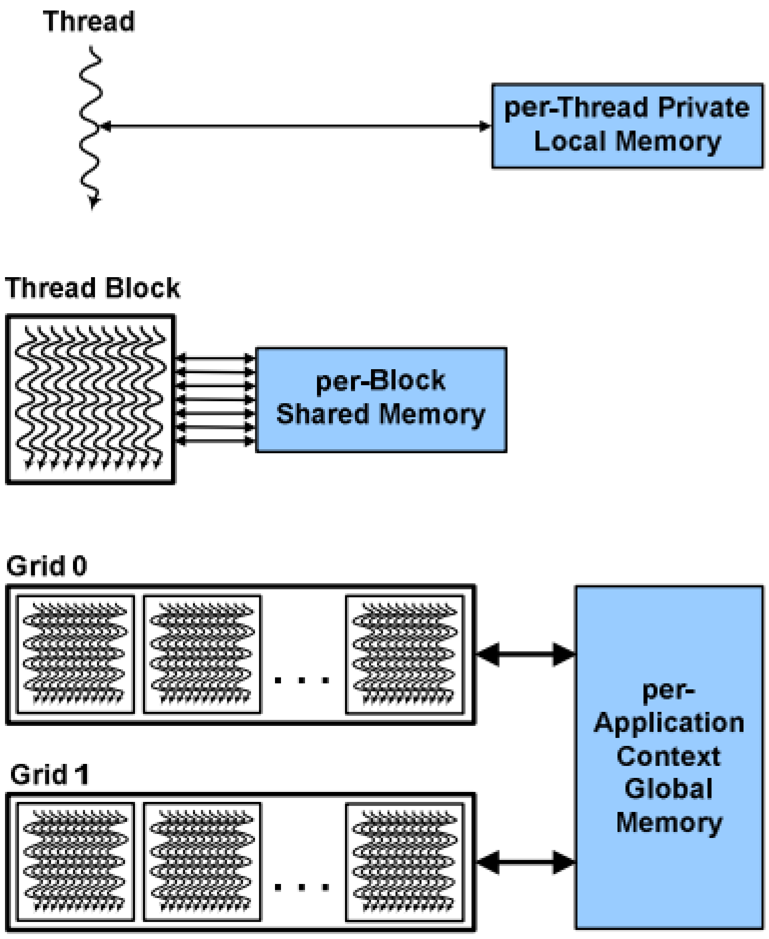

Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
. . .
per-Application Context Global Memory

Figure 2.11: CUDA Hierarchy of threads, blocks and grids with corresponding per-thread private, per-block shared, and per-application global memory spaces. Adapted from "NVIDIA Fermi Compute Architecture Whitepaper", by Nvidia Corporation, 2009, p. 6. Copyright 2009 by Nvidia Corporation

CHAPTER 3

RWPRAY

We have implemented Radio Wave Propagation Modeling using Ray Tracing with the aim of helping radar environment simulator implementation. While describing the study in this thesis, we will shortly refer the study as RWPRay, which is abbreviation of radio wave propagation using ray tracing. We have used ray tracing method to model the radio waves. Reflection, refraction and free space path loss are implemented within the scope of this project.

RWPRay has two functionalities: creating the propagation map, and finding position and power of all rays at which point they first time become closer to a specific target more than a given threshold region. Let us denominate these two functionalities as RWPRayMap and RWPRayTarget respectively. Figure 3.1 describes RWPRayTarget.

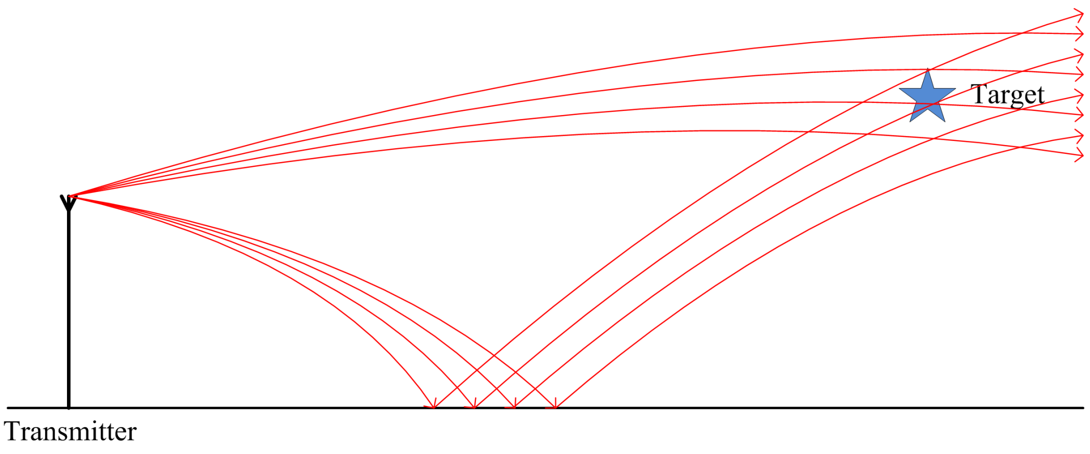


Figure 3.1: RWPRayTarget

RWPRay works on height map of terrains, including surface type map of the field. Propagation velocity change function of the region is selected by the user, and refractive indices are calculated by the application itself, according to user choices. They are stored in a table and this table is used during calculations. The table is created off line, thus different complexities of different propagation velocity change functions do not affect the performance and execution time.

The programming language that RWPRay implemented with is C++. GPU implementation is for Nvidia, and is developed using CUDA. Parallel CPU version of RWPRay is parallelized using multi-threading concepts.

RWPRay is implemented for GPU and CPU separately, and for CPU with two different versions, one of them is working sequential and the other is working parallel. Implementation is explained in Chapter 4 in detail.

3.1 Map Splitting

RWPRay works on 2D maps and creates results for them. However it is also capable of creating 2D slices of a 3D map and working on them. It is assumed that we can simulate radio wave propagation on 3D maps with creating 2D slices of them, modeling the propagation on created 2D maps and combining the 2D propagation maps to create the 3D one. On this work it is assumed that propagations on different slices does not affect each other. For creation of 2D slices, user defines the region of interest in horizontal and the slice resolution. Count of 2D maps what will be created is calculated as follows:

$$sc = \frac{\alpha_e - \alpha_s}{\Delta\alpha} \tag{3.1}$$

where:

sc **:** 2D map (slice) count

α_e **:** End angle of slices

α_s **:** Start angle of slices

$\Delta\alpha$ **:** Slice resolution

Angles of the slices that will be created is calculated as follows:

$$\alpha_k = \alpha_s + k \times \Delta\alpha \tag{3.2}$$

where:

α_k **:** Angle of k$^{\text{th}}$ slice

α_s **:** Start angle of slices

k **:** Slice number

$\Delta\alpha$ **:** Slice resolution

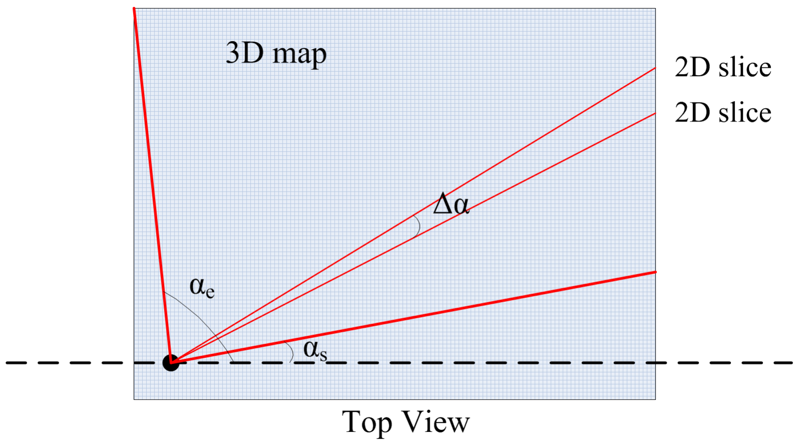


Figure 3.2: Map Splitting

Figure 3.2 shows how to create 2D slices from a 3D map.

3.2 Ray Tracing Calculation

While using RWPRay, the user defines position of the transmitter (radar), and the angles of the interval through which the rays will be sent, including the resolution. This interval describes the region of interest. Number of the rays is calculated according to the interval and resolution value. For instance, when the region of interest is from -1° to +1° and the resolution is 0.01°, 200 rays are created and calculated.

$$rc = \frac{\theta_e - \theta_s}{\Delta\theta} \tag{3.3}$$

where:

rc **:** Ray count

θ_e **:** Region of interest end angle

θ_s **:** Region of interest start angle

$\Delta\theta$ **:** Ray resolution

All rays are created by the transmitter, which is the radar in our system, from its position. Hence, first point of all rays are the same point which is the radar position. All rays have a power value and a direction. Initial power values of the rays are gotten from antenna pattern table of the radar depending on the angle, and initial angle is calculated according to user parameters which are region of interest start angle and resolution value. Refraction index of initial position is read from the table and stored as *previous index*.

$$\theta_k = \theta_0 + k \times \Delta\theta \tag{3.4}$$

where:

θ_k **:** Initial angle of k$^{\text{th}}$ ray

θ_0 **:** Region of interest start angle

k **:** Ray number

$\Delta\theta$ **:** Ray resolution

However, during the calculations, direction vector is used instead of the direction angle. Under favor of this, reflection and refraction are calculated using vector operations instead of trigonometric functions, which is a more efficient method. Direction vector is the unit vector of the ray direction. It is calculated as follows:

$$\vec{d} = \cos(\theta)\hat{\imath} + \sin(\theta)\hat{\jmath} \tag{3.5}$$

where:

$\vec{d}$ **:** Direction unit vector

θ **:** Initial angle of the ray

The ray is carried forward one step on x axis. X axis step length is the map resolution, which is the minimum distance for which it is assumed that the terrain height value is the same. Y axis step length is calculated using the unit vector as follows:

$$\Delta y = \frac{\Delta x j}{i} \tag{3.6}$$

where:

Δy **:** Y axis step length

Δx **:** X axis step length which is also the map resolution value

j **:** Y component of the unit vector

i **:** X component of the unit vector

Total step range is calculated using Pythagorean theorem as follows:

$$\Delta s = \sqrt{(\Delta x)^2 + (\Delta y)^2} \tag{3.7}$$

where:

Δs **:** Step range

Δx **:** X axis step length

Δy **:** Y axis step length

The next position of the ray is calculated adding found step values to the current position of the ray:

$$y_{\mathrm{k+1}} = y_{\mathrm{k}} + \Delta y \tag{3.8}$$

$$x_{\mathrm{k+1}} = x_{\mathrm{k}} + \Delta x \tag{3.9}$$

Firstly, new position of the ray is checked whether it is still in the map region or not. Then it is compared with height value of the terrain at that position. If the terrain height value is grater than or equal to the ray position, a collision occurs. When a ray hits the terrain, it reflects and continues along its path. The reflection is explained in Section 3.3 in detail.

If there is no collision, refraction index of new position of the ray is obtained from the table and named as *next index*. The ray is refracted using refraction indices, *next index* and *previous index*. Refraction process is explained in Section 3.4 in detail.

According to the step range, new power of the ray for this position is calculated. For this calculation, free space path loss formula is used. This process is explained in Section 3.5 in detail.

With the refraction, new direction vector is calculated. New position and new direction vector is stored as a new point of the ray. New refraction index is stored as previous one. After that the same process continues and the ray goes forward on its path.

While the ray is in the map region, there are two possibilities to terminate the ray. These are ray power and step count. This is selected by the user as *termination criteria*. If the user chooses the ray power, he also defines the termination power. When ray power is less than the termination power, this ray is regarded as terminated. When the step count is chosen as the termination criteria, maximum ray length (step count) is defined by the user. This time rays continue to their path until they reach the step limit. If the ray goes out of the map region, it terminates independently from the termination criteria.

When a ray is terminated, another waiting ray is calculated and so on.

3.3 Collision and Reflection

When height of the ray is less than or equal to the terrain height at a position, it is regarded as the ray hits the terrain. It reflects from the terrain and continues along its path. The ray changes its direction and lose power when such a collusion occurs.

3.3.1 Direction Calculation of Reflection

New direction of a ray after a reflection, is calculated with vectorial operations as follows [7]:

$$\vec{d} = \vec{I} - 2.0 \times dot(\vec{N}, \vec{I})\vec{N} \tag{3.10}$$

where:

$\vec{d}$ **:** New direction vector

$\vec{I}$ **:** Incident vector (current direction vector)

$\vec{N}$ **:** Normal vector (should be normalized)

$dot()$ **:** Dot product of two vectors

Normalized normal vector is calculated as follows:

$$diff = \frac{m_1 - m_0}{m_{res}} \tag{3.11}$$

$$N.j = \sqrt{\frac{1.0}{diff^2 + 1.0}} \tag{3.12}$$

$$N.i = -diff \times N.j \tag{3.13}$$

where:

m_1 **:** Height of terrain at the position the collision occurs

m_0 **:** Height of terrain at the previous position

m_{res} **:** Map resolution

$N.j$ **:** Y component of the normal vector

$N.i$ **:** X component of the normal vector

3.3.2 Power Calculation of Reflection

When a ray hits the terrain, the terrain absorbs some part of the ray power, and the ray continues along its path with less power [3, 1]. Each point of terrain has its surface type parameter and each surface type has it own dielectric and conductivity constants. Reflection coefficients of the surfaces are calculated by the system using these constants and radar parameters as follows:

$$v_0 = 120\pi \tag{3.14}$$

$$\epsilon_r = \epsilon + \frac{60\sigma c}{f} i \tag{3.15}$$

$$v_2 = \begin{cases} \frac{v_0\sqrt{\epsilon_r - 1.0}}{\epsilon_r}, & \text{if radar polarization is vertical} \\ v_0\sqrt{\epsilon_r - 1.0}, & \text{if radar polarization is horizontal} \end{cases} \tag{3.16}$$

$$r = \left| \frac{v_2 - v_0}{v_2 + v_0} \right| \tag{3.17}$$

where:

ϵ **:** Dielectric constant of the surface

σ **:** Conductivity constant of the surface

c **:** Speed of light

i **:** $\sqrt{-1}$ unit imaginary number

f **:** Radar frequency

r **:** Reflection coefficient

In our system we use five different predefined surface types [14]. They are listed with their dielectric and conductivity constants in Table 3.1.

Number of bound charges in a material is a metric that is called the permittivity. Permittivity is expressed as a multiple of the permittivity of free space, ϵ_0. This term is called the relative permittivity, ϵ, or the *dielectric constant* of the material.

Conductor materials are characterized by their conductivity, σ. *Conductivity* is ability of a material to conduct an electric current. Most real-world materials will have both a dielectric constant and a nonzero conductivity. As the conductivity of the dielectric material increased, the dielectric becomes more lossy.

Table 3.1: Surface Types

Surface Type	Dielectric Constant (ϵ)	Conductivity Constant (σ)
Sea	69.1342	7.1462
Fresh water	80.0	1.4978
Wet ground	25.9161	0.67002
Medium fry ground	15.0	0.22934
Very dry ground	3.0	0.0023007

Apart from these, the user can define a new surface type and define its dielectric and conductivity constants. The system can execute with user defined surfaces as well.

3.4 Refractive Index Calculation and Refraction

In RWPRay, rays refract at each step according to the refraction indices of the previous and current positions of the ray. Refraction changes the direction of the ray but not the power. It is calculated with vectorial equations as follows [8]:

$$k = 1.0 - \eta^2(1.0 - dot(\vec{N}, \vec{I})^2) \tag{3.18}$$

$$\vec{d} = \begin{cases} 0.0, & \text{if k < 0.0} \\ \eta\vec{I} - (\eta \times dot(\vec{N}, \vec{I}) + \sqrt{k}), & \text{otherwise} \end{cases} \tag{3.19}$$

where:

$\vec{d}$ **:** New direction vector

$\vec{I}$ **:** Incident vector (should be normalized)

$\vec{N}$ **:** Normal vector (should be normalized)

η **:** Ratio of indices of refraction

$dot()$ **:** Dot product of two vectors

Rays are refracted by horizontal layers. Thus, normal vector of refraction is defined as follows:

$$\vec{N} = (0, -1) \text{ when the ray has a positive y direction} \tag{3.20}$$

$$\vec{N} = (0, 1) \text{ when the ray has a negative y direction} \tag{3.21}$$

Refraction indices are assumed to be changed every meter for high accuracy. They are calculated for every meter and stored in a table to be used during the calculations. User preferences specify how to calculate the refraction indices. Before the refraction index change function, the user selects whether these constants change with the range or not.

In range independent choice, it is assumed that, refractive index does not change at the same hight with a changing range. Each altitude value, with 1 meter resolution, has its refractive index and it is constant at all points of the map. The atmosphere consists of one-meter-height horizontal layers in which the propagation velocity is constant. The user defines only one change function and this applies for the whole map.

In range dependent choice, it is assumed that, there are range limits which make the refractive index change function shift by another one. Each altitude value still has its own refraction index, however it may change at user defined range limits. In this method, the user first defines the range limits where refractive index calculations will shift, and then he matches one change function to each range region.

There are 5 different atmosphere types that have different refractive index change functions [14, 21, 17].

3.4.1 Standard Atmosphere

In standard atmosphere, there is linear dependency between the refractive index and the altitude. Slope of the modified refractivity is constant and can be used to calculate index value of any height. In this type, user defines the modified refractive index for 0 altitude and any other point giving its height. The system creates the table for all possible altitude values in selected map using the given modified refractive indices. Figure 3.3 describes the standard atmosphere method.

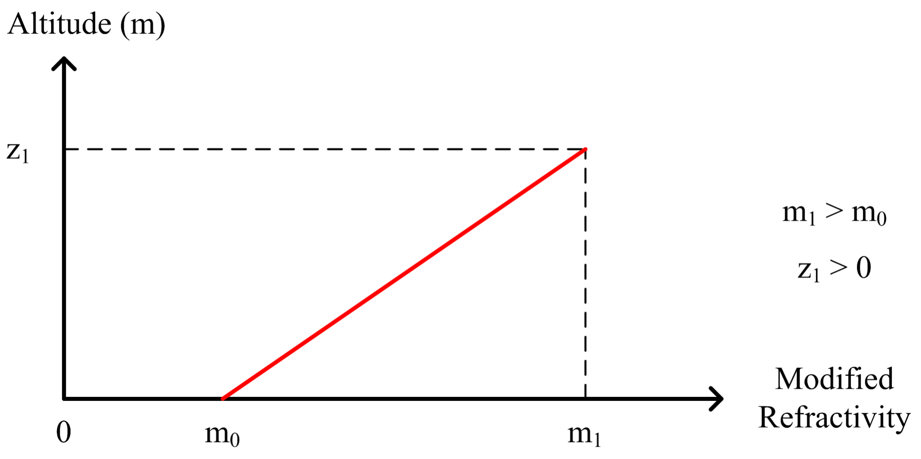


Figure 3.3: Standard Atmosphere

3.4.2 Surface Duct

In surface duct, refractive index still changes linearly. However, there is an altitude value where the slope of the modified refractivity changes. In that type, user defines modified refractive index of two more points apart from the 0 altitude point. The system creates the table according the user input for all possible altitude values in selected map. Figure 3.4 describes the surface duct method.

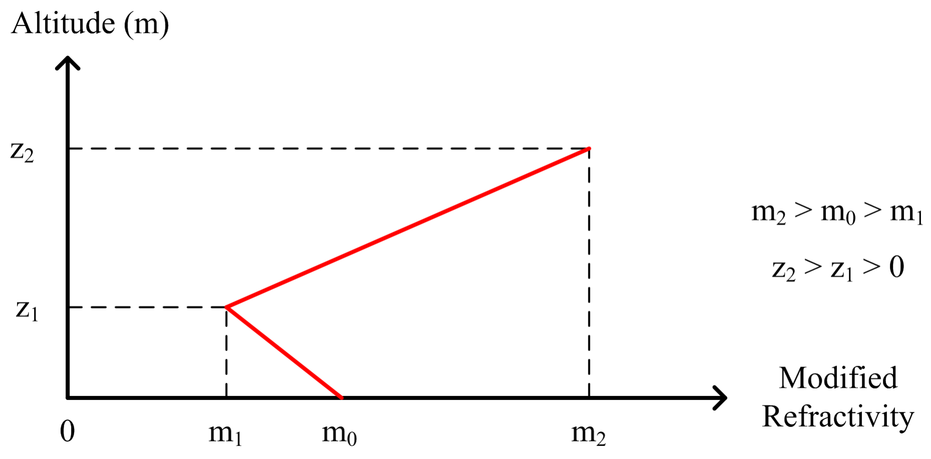


Figure 3.4: Surface Duct

3.4.3 Surface Based Duct

In surface based duct, refractive index changes linearly, besides, there are two altitude values where the refractive index changes its behavior. In that type, apart from the 0 altitude point, the user should define three points and modified refractive indices of them. Which is important in here is that, modified refractive index of second highest point should be less than the 0 altitude point's one. The system creates the table according to all four points for all possible altitude values. Figure 3.5 describes the surface based duct method.

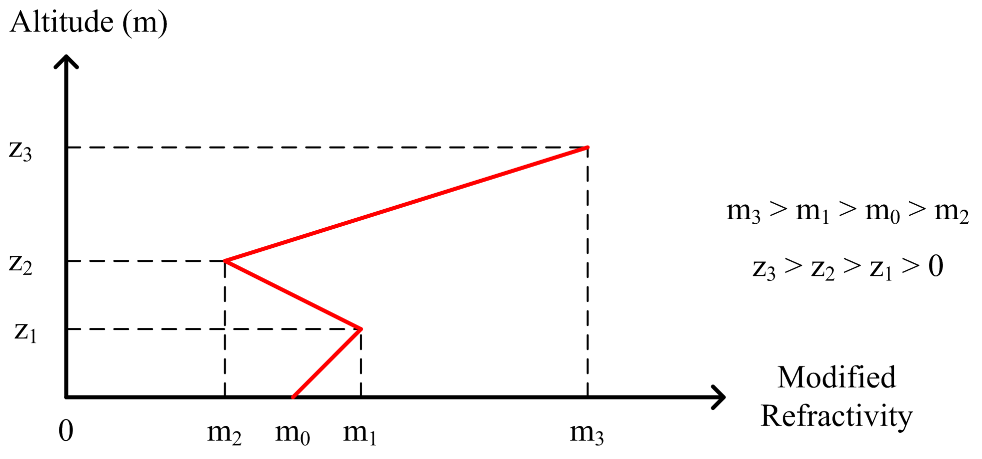


Figure 3.5: Surface Based Duct

3.4.4 Elevated Duct

Elevated duct is very similar to surface based duct but it has only one difference. In elevated duct, modified refractive index of 0 altitude point should be the lowest one. Figure 3.6 describes the elevated duct method.

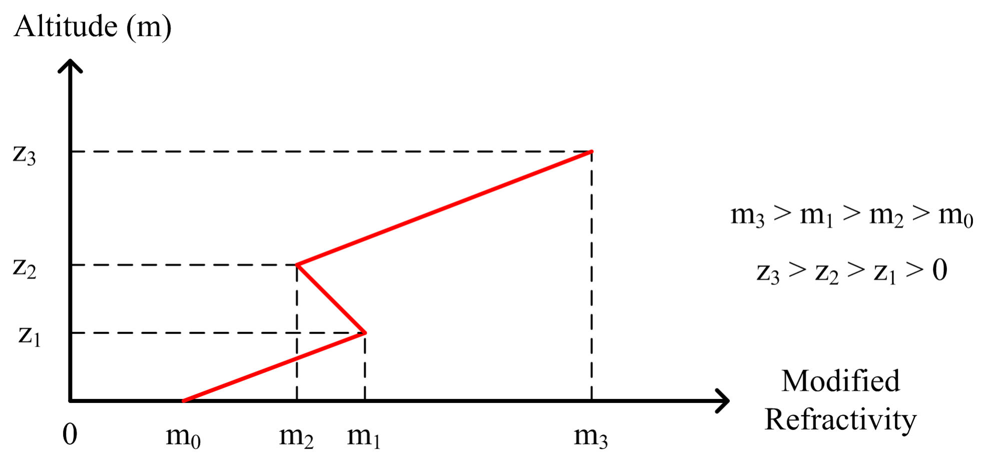


Figure 3.6: Elavated Duct

3.4.5 Evaporation Duct

Evaporation duct is different from all other methods because in evaporation duct the refractive index changes parabolically. In this method, user defines modified refractive index of 0 altitude point and the duct height. The system creates the table for all possible altitude values according to user parameters. Figure 3.7 describes the evaporation duct method.

3.5 Attenuation

Rays lose power during the travel in the atmosphere. This is also called attenuation. The power of the ray at a specified point is inversely proportional to the square of distance between this point and the transmitter. When the initial power of the ray is given, the power at any point is calculated with free space path loss equation as follows:

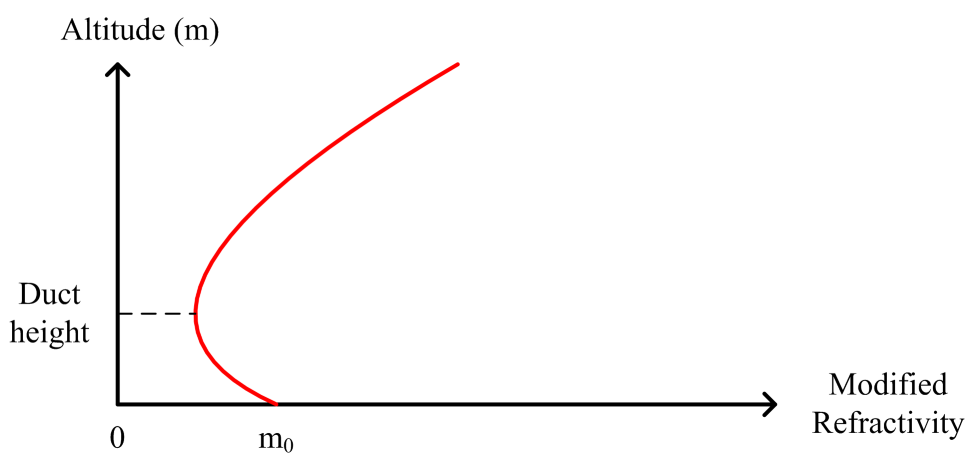


Figure 3.7: Evaporation Duct

$$P_1 = \frac{P_0}{r^2} \tag{3.22}$$

where:

P_1 **:** Ray power at a specified point

P_0 **:** Initial ray power

r **:** Range of the specified point (the total path that the ray has passed through)

During the calculation, if the initial power of the ray is not known, but the power at any point is known, the power of a different point is calculated as follows:

$$P_2 = \frac{P_1 r_1^2}{r_2^2} \tag{3.23}$$

where:

P_2 **:** Ray power at 2nd point

P_1 **:** Ray power at 1st point

r_2 **:** Range of the 2nd point (the total path that the ray has passed through)

r_1 **:** Range of the 1st point (the total path that the ray has passed through)

CHAPTER 4

IMPLEMENTATION

We have developed three different versions of RWPRay which are running sequential on CPU, parallel on CPU and parallel on GPU. For all floating point variables, 8 byte doubles are used. Even though 4 byte floats give better performance, usage of 4 byte floats caused the anomaly of broken rays because of the low precision of 4 byte floats.

At first, we stored points of the rays with direction angle. We performed the reflection, refraction and step calculations using this direction angle and trigonometric functions. Then we made an improvement by storing the direction unit vector instead of the angle. We reimplemented the reflection, refraction and step calculations using the direction vector, and vector operations. This modification brought us a performance improvement of about 40% on CPU implementations and about 10% on GPU implementation.

Moreover, in current version of RWPRay, refractive indices are calculated during initialization and stored in a table. They are read from the table during the calculations. We have tried to calculate the refractive indices during the ray calculations and tried not to use the table. We have implemented and tested it with surface duct atmosphere type, which has a medium complexity among other surface types. We could not achieve any performance gain with this approach. Therefore, we concluded that refractive indices that are calculated offline gives the best performance.

RWPRay results are verified using the implementation provided in [17].

All three versions of RWPRay accept these shared parameters:

mapName : Name of the input map file

startAngle : Start angle of region of interest

endAngle : End angle of region of interest

rayResolution : Resolution of rays in region of interest

terminationCriteria : Termination criteria of the rays (power or step count)

stepCnt : Termination step count of the rays

minPower : Termination power of the rays

atmosphereType : Atmosphere type that is defined for the map

atmosphereParams : Parameters of the specified atmosphere type to calculate refractive indices table

radarFreq : Frequency of the radar signal

polarization : Radar polarization (horizontal or vertical)

radarPosition : Position of the radar in the map

When RWPRay work on 3D maps, following parameters are also needed:

sliceStartAngle : Start angle of region of interest on horizontal

sliceEndAngle : End angle of region of interest on horizontal

sliceResolution : Resolution of slices in region of interest on horizontal

According to all user parameters, RWPRay calculates the rays in the system one by one and stores them. At the end of a successful run, RWPRay creates two output files, one of them is for logs and the other is for output. Log file is for working summary. It includes the parameters and the execution time. The execution time is measured for only the ray tracing part. Reading the parameters and doing the initializations are not measured and not included into the time statistics because we did not do any study on that part and it is exactly the same on the all three versions. Resultant rays are stored in the output file. Both of the files are decided to be text files for ease of handling.

4.1 Sequential on CPU

This version of RWPRay works completely sequential, in only one thread which is the main thread. It gets the user parameters, reads the map file, creates the refractive indices table, and starts to do the radio wave propagation modeling in sequence. If it is chosen to work on a 3D map, 2D map slices are also created by only one thread in sequential manner. After all rays are calculated, RWPRay writes the results to log and output files and exits.

4.2 Parallel on CPU

This version of RWPRay works in multi threading concepts. It does the initializations in sequential, then creates threads to calculate the rays. Getting the user parameters, reading the map file and creation of the refractive indices table are done in the main thread sequentially. Then, main thread creates the other threads and passes them the parameters. Each worker thread calculates the rays assigned to it. The thread that will calculate a ray depends on the ray number and thread count. It is found as follows:

$$tId = rId \,\%\, tCnt \tag{4.1}$$

where:

tId **:** Id of the thread that will calculate the ray with the id *rId*

rId **:** Id of the ray that will be calculated

$tCnt$ **:** Total worker thread count

$\%$ **:** Modulo operation

After worker threads finish calculation of the rays, main thread writes the results to the files and it exits.

4.3 Parallel on GPU

GPU version of RWPRay is implemented with C++, using CUDA, for nvidia graphics cards. Reading the input and doing the initializations are done on the CPU. Ray calculations and creating the 2D slices from 3D maps are done on the GPU. The system gets user parameters, reads the map file and creates the refractive indices table according to the user parameters. It prepares the texture memory and copies the map to the texture memory. Then it allocates places for needed variables on the GPU memory and moves the parameters other than the map to there. Memory for the output is also allocated from the GPU.

Map is stored in texture memory to increase the performance. Texture memory is a read only memory and its speed is greater than the global memory.

After this preparation, CUDA kernel is called. All ray calculations are done in the CUDA kernel function. Kernel functions return asynchronously. After this function returns, the system waits for all CUDA threads finish. Each CUDA thread calculates the rays that are reserved for it, and writes the result to the place that is allocated for that calculated ray. When the thread finishes all rays that are under its responsibility, it returns. Each thread chooses its rays using following *for loop*:

```
for( int rayId = tId; rayId < rayCnt; rayId = rayId + tCnt )
```

where:

rayId : Id of the ray that will be calculated

tId : Id of the running CUDA thread

rayCnt : Total ray count

tCnt : Total CUDA thread count

After all rays finish, CPU code is informed and it continues to run. It transfer the output from GPU memory to system memory, and writes them to files. Then the whole program exits.

CHAPTER 5

RESULTS

5.1 Test Environment

Running time measurements of CPU implementations (sequential and parallel) are done on two different CPUs. These implementations are compiled both with and without optimization option (O2), and performance of the both are analyzed. Results of unoptimized compilations are not covered in this thesis. Running time results are presented and analyzed in Section 5.4. Parallel implementation is run in 16 threads apart from the main thread, and sequential implementation is run in the main thread. Specifications of these CPUs are given in Table 5.1.

Table 5.1: CPU Specifications

	CPU1	**CPU2**
Computer Model	HP Z200 Workstation	HP Z820 Workstation
Processor Model	Intel Xeon X3440	2 $\times$ Intel Xeon E5-2690
# of Cores	4	2 $\times$ 8
Clock Speed	2.53 GHz	2.90 GHz
Intel Smart Cache	8 MB	20 MB
Instruction Set Extensions	SSE4.2	AVX
RAM	4 GB	32 GB

Running time measurements of GPU implementation are also done on two different nvidia graphics cards. Pure process times and memory operation included process times are saved separately and they are both analyzed in Section 5.4. GPU implementation is run with 32 CUDA blocks and 512 CUDA threads per each block. Specifications of these GPUs are given in Table 5.2.

Table 5.2: GPU Specifications

	GPU1	**GPU2**
Computer Model	HP Z820 Workstation	HP Z820 Workstation
Card Model	Nvidia Tesla C2075	Nvidia Tesla K20
Type of GPU	Tesla T20A	Kepler GK110
Architecture	Fermi	Kepler
# of CUDA Cores	448	2496
Frequency of CUDA Cores	1.15 GHz	706 MHz
Peak Double Precision Floating Point Performance	515 Gflops	1.17 Tflops
Peak Single Precision Floating Point Performance	1.03 Tflops	3.52 Tflops
Total Memory	6 GB GDDR5	5 GB GDDR5
Memory Clock Frequency	1.50 GHz	2.6 GHz
Memory Bandwidth	144 GB/sec	208 GB/sec

For the running time measurements, we have run all the tests 50 times and took the average times. Minimum and maximum durations are excluded from the average calculation.

All running time measurement values may be downloaded from `http://www.ceng.metu.edu.tr/~e1449297/thesis/allTimeVals.zip`

RWPRayMap gives the propagation map as output. The propagation map is formed of calculated rays. Each ray consists of ray points connected to each other. Ray points are composed of the following variables:

- X position
- Y position
- Total range
- Ray power

When ray points and rays combine and constitute the propagation map, it comes up with a huge data for especially big maps and wide regions of interest. This situation causes long memory operation times on GPU implementation.

RWPRayTarget finds all rays' closest point to a specified target. It stores only 1 point for each ray. Under favor of that, the output data is significantly less than RWPRayMap version. RWPRayTarget version gives time results that the memory operation times decrease to ignorable levels on GPU implementation.

All tests are done using the same atmosphere parameters. Changing these parameters does not affect the performance because refraction indices are calculated using the atmosphere parameters and are stored in a table. During the ray calculations, RWPRay reads refraction indices from this table. Thus, complexity of the refraction indices calculation does not change the execution time. Atmosphere parameters that are used in the experiments are as follows:

atmosphere type : Surface duct

range start : 0 m

range end : Map range

frequency : Radar frequency

number of elements : 3

altitude value 1 (z_0) **:** 0 m

modified refractive index 1 (m_0) **:** 350

altitude value 2 (z_1) **:** 800 m

modified refractive index 2 (m_1) **:** 300

altitude value 3 (z_2) **:** 1200 m

modified refractive index 3 (m_2) **:** 275

5.2 Data Sets

We have done the time measurements on 3 different maps and with 3 different ray sets. In total, we have analyzed timing results of 9 different runs. Map1 and map2 have the same range but different resolutions, hence they have unequal sizes. Map3 has the highest resolution and size. It also has the highest range.

All maps are all-zero altitude, flat fields. The terrain affects the rays and it may cause some of the rays to leave the map earlier. Because the ray counts differ from test to test, this situation may create unfair cases among the maps. Therefore, we decided to do the time measurements with all-zero fields to try to avoid potential unfair situations.

Ray sets differ only with the ray count from each other. They all have the same start angle and ray resolution. Of course, different ray counts with the same resolution results in different end angles.

5.2.1 Maps

In this study, we did the tests using three different maps. Details of these maps are given in this section.

Table 5.3: Specifications of the Maps

	Map 1	Map 2	Map 3
Size	300	3000	10000
Resolution (m)	1000	100	50
Range (m)	300000	300000	500000
Max Altitude (m)	10000	10000	10000

Map1 has the lowest resolution. Thus step sizes of the rays for this map are greater than the other maps'. This situation causes to get the least accurate results in shortest time from this map.

Map2 has the same range with map1. However map2 has a higher resolution with greater size than map1. This causes to get more accurate results in a longer time duration.

Map3 is the map that has the highest range with both higher resolution and greater size. Map3 is the map that gives the most accurate results because of its high resolution. It also consumes the longest time because of its size.

Specifications of all three maps are given in Table 5.3.

5.2.2 Ray Sets

In this study, we did the tests using three different ray sets. Details of these ray sets are given in this section.

Table 5.4: Specifications of the Ray Sets

	Ray Set 1	Ray Set 2	Ray Set 3
Start Angle (°)	-2.0	-2.0	-2.0
End Angle (°)	-1.96	-1.6	2.0
Spanned Area (°)	0.04	0.4	4.0
Resolution (°)	0.0004	0.0004	0.0004
Ray Count	100	1000	10000

Ray Set 1 has the least ray count. It spans the smallest area, with 0.04°, among the other ray sets.

Ray Set 2 spans an area with 0.4°, which is the median in our tests.

Ray Set 3 has the highest ray count and it spans the largest area, with 4.0°. This set is the most challenging one we analyzed in this study.

Specifications of all three ray sets are given in Table 5.4.

5.3 Validation

We have validated our results using Sevgi's ray-shooting Matlab package [17]. We consider the comparison results of two of our tests in this section.

The first test is the one that uses map 1 and ray set 1. Figure 5.1 shows comparison of RWPRay's result with the result of Sevgi's study. RWPRay's result is plotted in the first part and Sevgi's result is plotted in the second part. The third part shows the element-wise difference between two results for each ray. Absolute value of this difference is 1 on few places and 0 on the others. Ray positions are calculated using 8 byte doubles. However, they are exported and stored as integers. Because the results

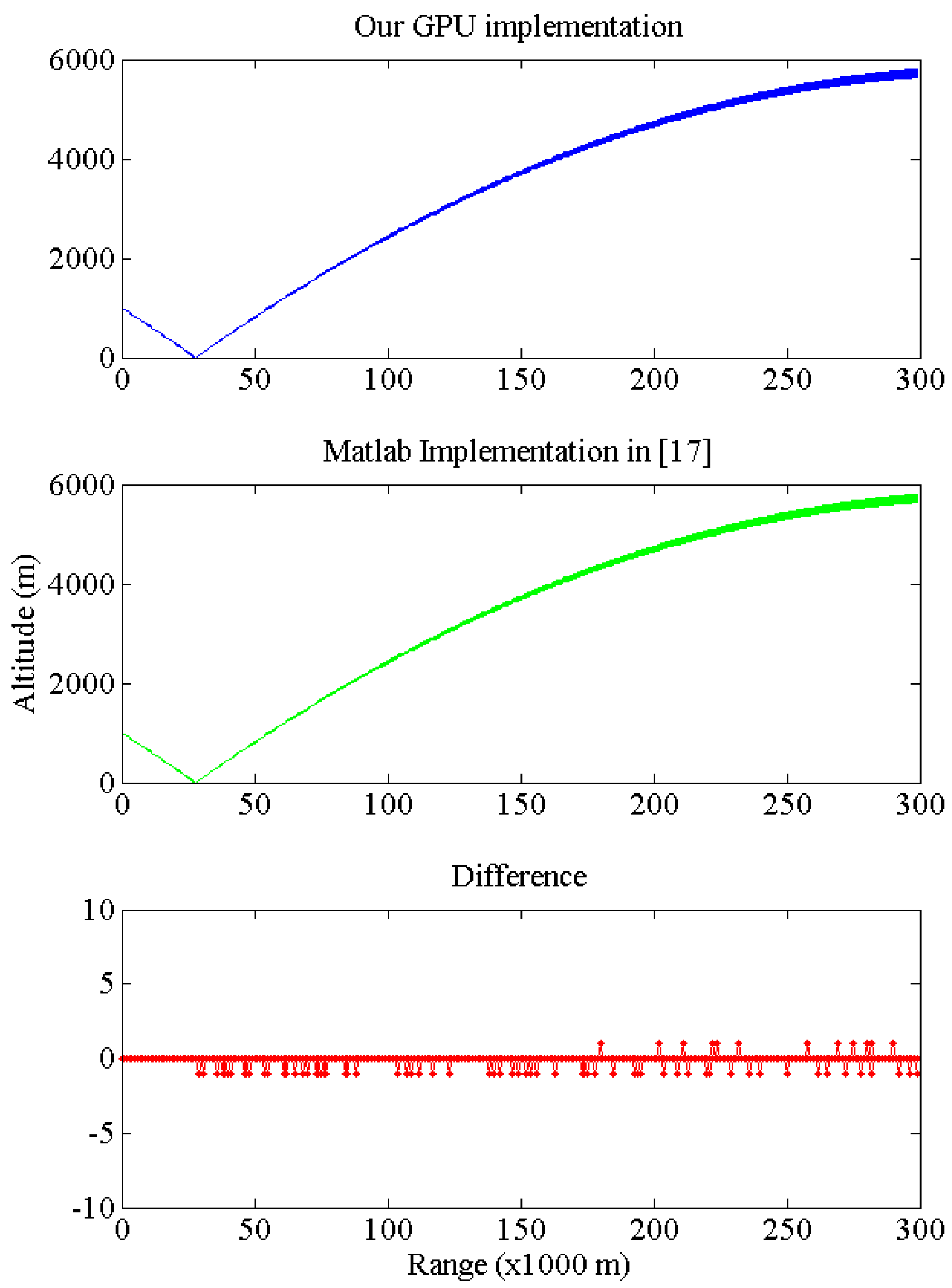


Figure 5.1: Differences for Map1 & RaySet1

are compared as integers, the differences are also integers. This double to integer conversion may end up with two possible results. The first one, it may hide some less-than-one differences and make them invisible to us. The second one, it may round out some less-than-one differences to one and highlight them. Now we are sure there are no difference greater than one in this comparison. Maximum height of the rays is about 6000 m and this results in 1/6000 error rate, which is less than 0.017%. This can be interpreted as 1 meter error in radar's range measurement and this error rate is acceptable for the radars considered in this study.

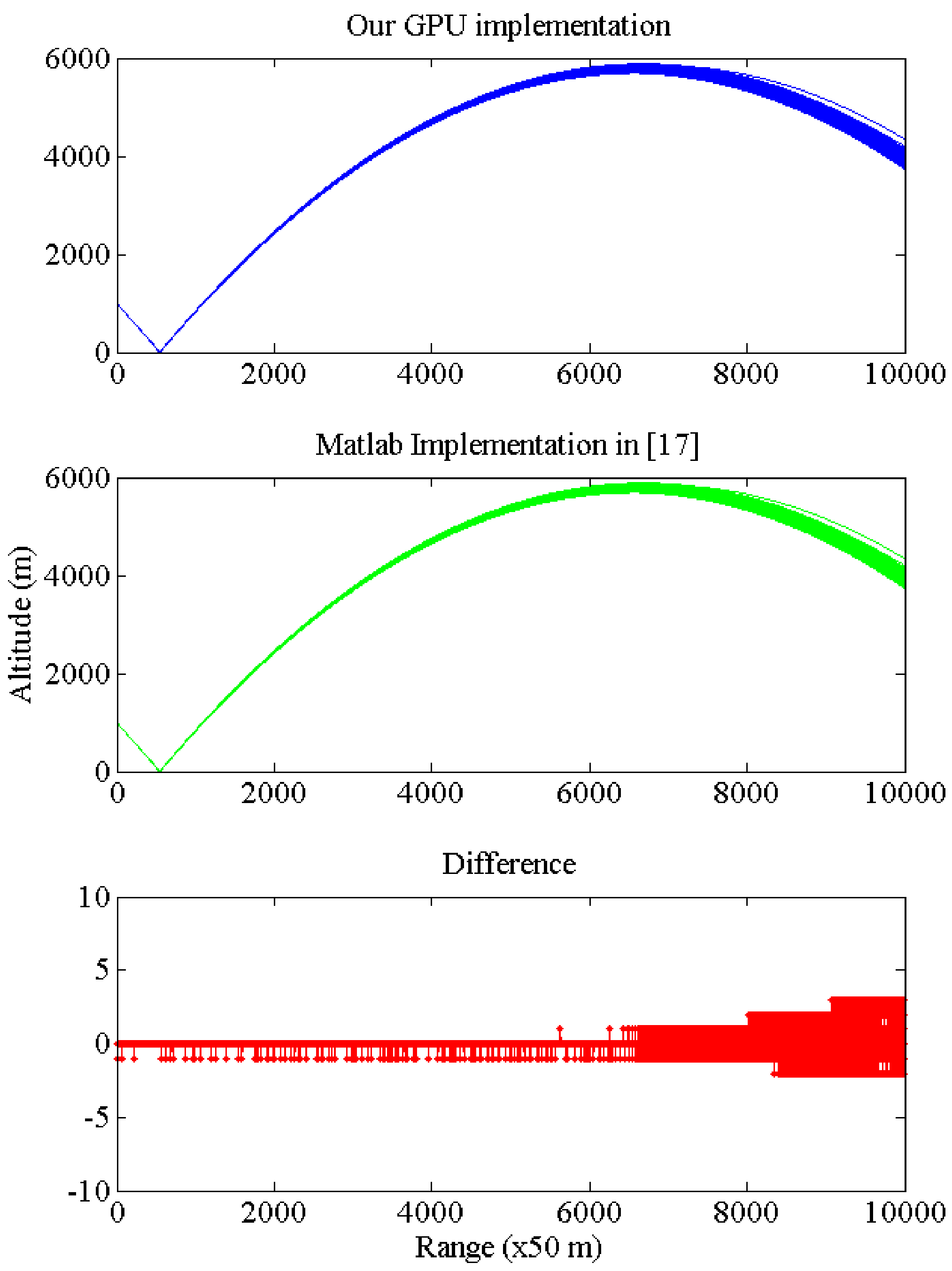


Figure 5.2: Differences for Map3 & RaySet1

The second test that we consider in this section uses map 3 and ray set 1. Figure 5.2 shows comparison of RWPRay's result with the result of Sevgi's study. RWPRay's result is plotted in the first part and Sevgi's result is plotted in the second part. The element-wise difference between two results for each ray is plotted again in the third part. Absolute value of the difference is at most 1, until a range of 8000$\times$50 m. After that range, until the end of the map, which has a range of 10000$\times$50 m, absolute value of the difference is at most 3. In this test, rays change direction at a range about 6500$\times$50 m and the difference density increases after that point. Possible reason of that situation may be the difference in refraction calculation part. In our study, ray direction is stored as a vector and refraction is calculated using vectorial operations. In Sevgi's Matlab package, ray direction is stored with the angle and refraction is calculated using trigonometric functions. This difference may be the one that causes the results difference get a higher density after the rays change direction. At the maximum error density region, the height of the rays is about 4000 m and this results in 3/4000 error rate, which is 0.075%. This can be interpreted as 3 meter error in radar's range measurement and this error rate is acceptable for the radars considered in this study.

5.4 Test Results

This section shows resultant radio wave propagation maps of 9 different tests. Running time measurements of these 9 tests, for both RWPRayMap and RWPRayTarget are shown in tables separately, and in Section 5.4.10, all together. All time values are in **milliseconds**. Column titles, which are the same for all the tables are explained below:

Serial : Calculations are done in main thread

Parallel : Calculations are done in 16 threads apart from the main thread

Pure : Pure calculation time for the GPUs

Full : Calculation time including the memory operations for the GPUs

Speedup : Speedup between CPU and GPU (pure time)

Table 5.5 shows map and ray set of each test.

Table 5.5: Map and RaySet of Each Test

	Ray Set 1	Ray Set 2	Ray Set 3
Map 1	Test-1-1	Test-1-2	Test-1-3
Map 2	Test-2-1	Test-2-2	Test-2-3
Map 3	Test-3-1	Test-3-2	Test-3-3

5.4.1 Test-1-1

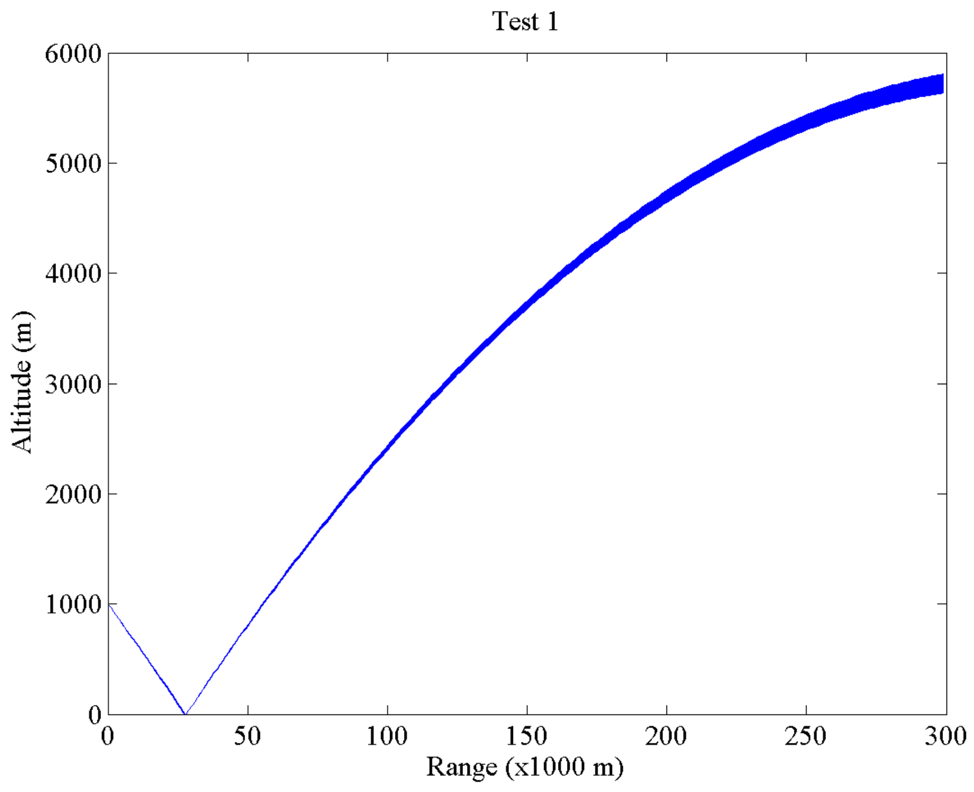


Figure 5.3: Map1 & RaySet1

Resultant radio wave propagation map of Test-1-1 is shown in Figure 5.3. It is spanning only an area of 0.04°, thus it is very narrow in places near the transmitter, and a little bit wider in further places. Rays are reflecting from the terrain and they are traveling not linear but curvedly by reason of the refraction.

Running time measurements of RWPRayMap for Test-1-1 is shown in Table 5.6 and RWPRayTarget's one in Table 5.7. We can see that CPUs are giving better perfor-

Table 5.6: RWPRayMap Times of Test-1-1

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
2.887	1.719	2.399	1.475	3.166	3.495	4.403	4.707

Table 5.7: RWPRayTarget Times of Test-1-1

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
3.113	1.627	2.401	1.411	3.409	3.440	4.587	4.625

mance than the GPUs for this test. Keep that in mind Test-1-1 is the lightest test because of both the map and the ray set, which are Map1 and RS1. If we compare RWPRayMap and RWPRayTarget, in the second one, dominance of the CPU increases. However, the difference between full and pure times of the GPU decreases. It is not clear enough in that example, but we expect to see this change in larger data sets clearly.

Performance sorting for test-1-1:
CPU2p > CPU1p > CPU2s > CPU1s > GPU1 > GPU2

Table 5.8: RWPRayMap Speedup for Test-1-1

	GPU1		GPU2	
	Serial	Parallel	Serial	Parallel
CPU1	0.91	0.54	0.66	0.39
CPU2	0.76	0.47	0.54	0.33

All speedup values are less than 1 and GPU1 has the greater speedups.

5.4.2 Test-1-2

Result map of Test-1-2 is shown in Figure 5.4. RS2 is the ray set that is spanning 0.4°, which is grater than the previous one. The spanned area is expanding while moving away from the transmitter and it is in a curved shape as we expect.

Running time measurements of RWPRayMap for Test-1-2 is shown in Table 5.9 and

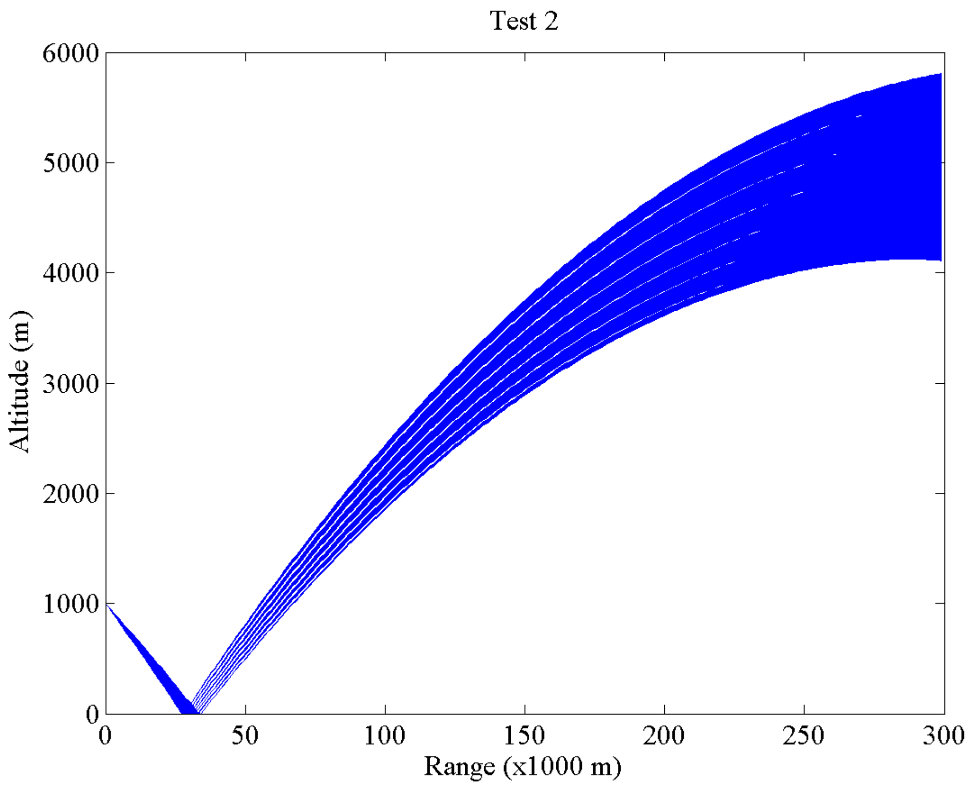


Figure 5.4: Map1 & RaySet2

Table 5.9: RWPRayMap Times of Test-1-2

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
26.674	7.623	19.280	2.781	4.732	6.857	5.066	6.920

Table 5.10: RWPRayTarget Times of Test-1-2

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
27.059	7.727	19.931	2.749	5.085	5.133	5.199	5.251

RWPRayTarget's one in Table 5.10. With a bigger ray set, which is RS2, GPUs are starting to give better performance than the CPUs. We could achieve 4.07 speedup at Test-1-2 between the faster GPU and faster CPU. When we look at the parallel implementations, while the faster CPU is still keeping ahead, the slower one slipped down of the GPUs. That is what we expect while the ray set is getting larger. Moreover, if we compare RWPRayMap and RWPRayTarget, the change in full-pure time difference is getting more noticeable.

Performance sorting for test-1-2:

CPU2p > GPU1 > GPU2 > CPU1p > CPU2s > CPU1s

Table 5.11: RWPRayMap Speedup for Test-1-2

	GPU1		GPU2	
	Serial	Parallel	Serial	Parallel
CPU1	5.64	1.61	5.27	1.50
CPU2	4.07	0.59	3.81	0.55

Only the parallel implementation running on CPU2 has a speedup less than 1 and GPU1 has the greater speedups. Moreover, the highest speedup is 5.64.

5.4.3 Test-1-3

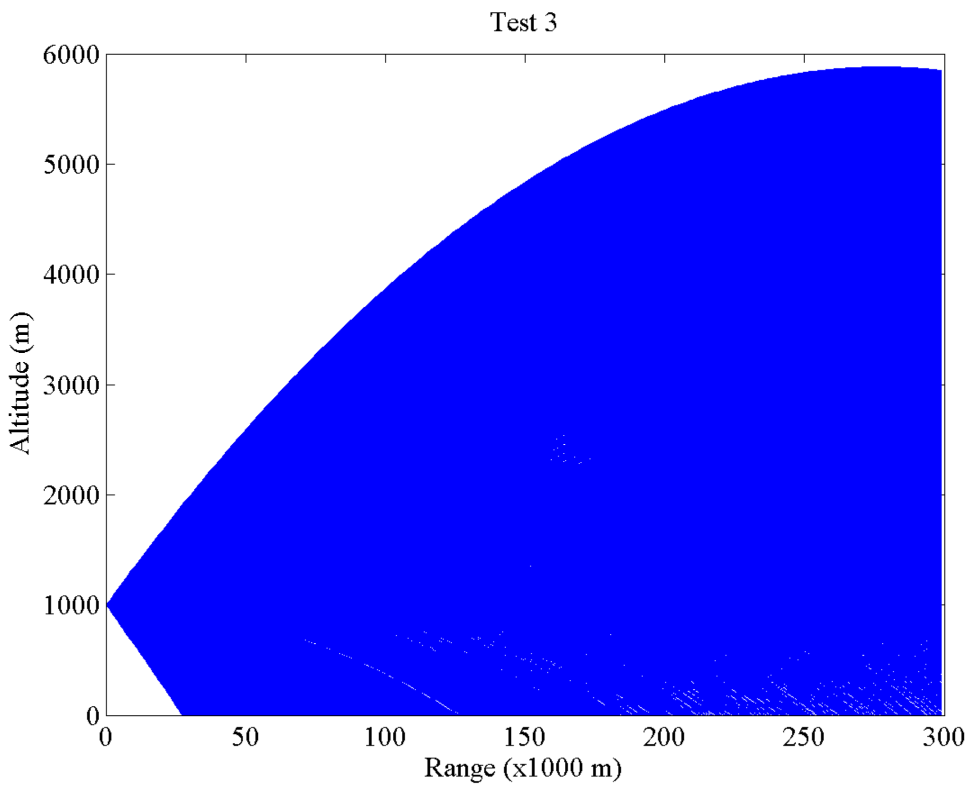


Figure 5.5: Map1 & RaySet3

Propagation map of test-1-3 is shown in Figure 5.5. Rays are spanning an area of 4°. Because this map is too crowded and it is hard to identify the rays, we are also showing a reduced version of it in Figure 5.6. In that reduced figure, instead of all

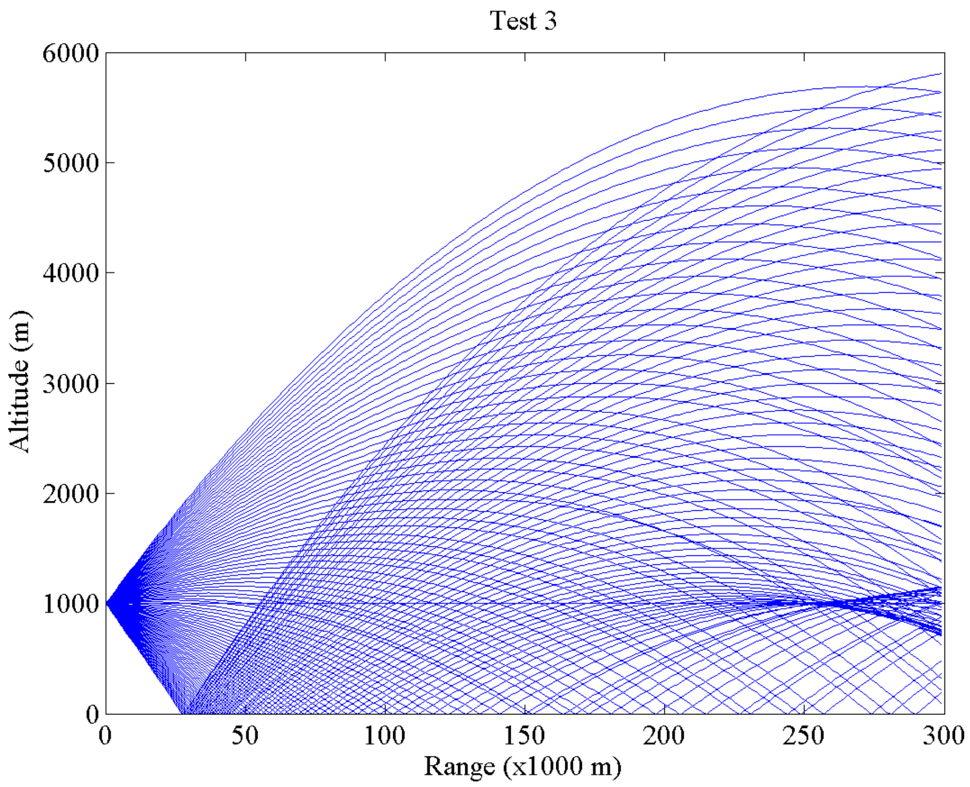


Figure 5.6: Map1 & RaySet3 (Reduced)

rays, every 100^{th} ray is plotted. It is more useful to pursue paths of the rays. They are reflecting from the terrain and following a curved path as we expect. Some of the rays are reflecting twice in the map region. This is possible and also expected by force of the refractivity for some but not all region of angles.

Table 5.12: RWPRayMap Times of Test-1-3

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
258.80	70.48	188.58	18.43	15.75	34.97	10.39	27.70

Table 5.13: RWPRayTarget Times of Test-1-3

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
268.12	70.82	185.15	18.38	15.63	15.84	10.61	10.80

RWPRayMap and RWPRayTarget running times are given in Tables 5.12 and 5.13 respectively. This test is done with the lightest map but the hardest ray set. Performances of the both GPUs are defeating the serial implementation on the both CPUs. Moreover, the GPUs are still faster than even parallel implementation on both CPUs by a narrow margin. Note that, in the most crowded ray set, the better GPU is 18 times faster than the better CPU.

However, which is the better GPU? In test-1-1 and test-1-2 GPU1 was the better, but in test-1-3 GPU2 is the faster one. Also note that, in the most crowded ray set, GPU2 outperforms GPU1.

When the ray count increases, memory operation time of the GPU also increases proportional to it. In RWPRayMap, there is a memory copy time even more than the process time. RWPRayTarget relieves us of that long time. Its contribution to the full time values is flashy now.

Performance sorting for test-1-3:
GPU2 > GPU1 > CPU2p > CPU1p > CPU2s > CPU1s

Table 5.14: RWPRayMap Speedup for Test-1-3

	GPU1		GPU2	
	Serial	Parallel	Serial	Parallel
CPU1	16.43	4.47	24.91	6.78
CPU2	11.97	1.17	18.14	1.77

All speedup values are greater than 1. This means for all possible matchings the GPU part will win. GPU2 has the greater speedups. Moreover 24.91 is the highest speedup.

5.4.4 Test-2-1

Radio wave propagation map that is created with test-2-1 is shown in Figure 5.7. This test is run with the lightest ray set and is spanning an area of 0.04°. Map of the test is map2, which shows the same filed with map1, but in a 10 times higher resolution. With this test, we procured 10 times more data than test-1-1, for the same field. As we are used to see, rays are reflecting and following a curved path in the field.

Running time values of RWPRayMap are shown in Table 5.15 and running time values of RWPRayTarget are shown in Table 5.16. According to these tables, test-2-1

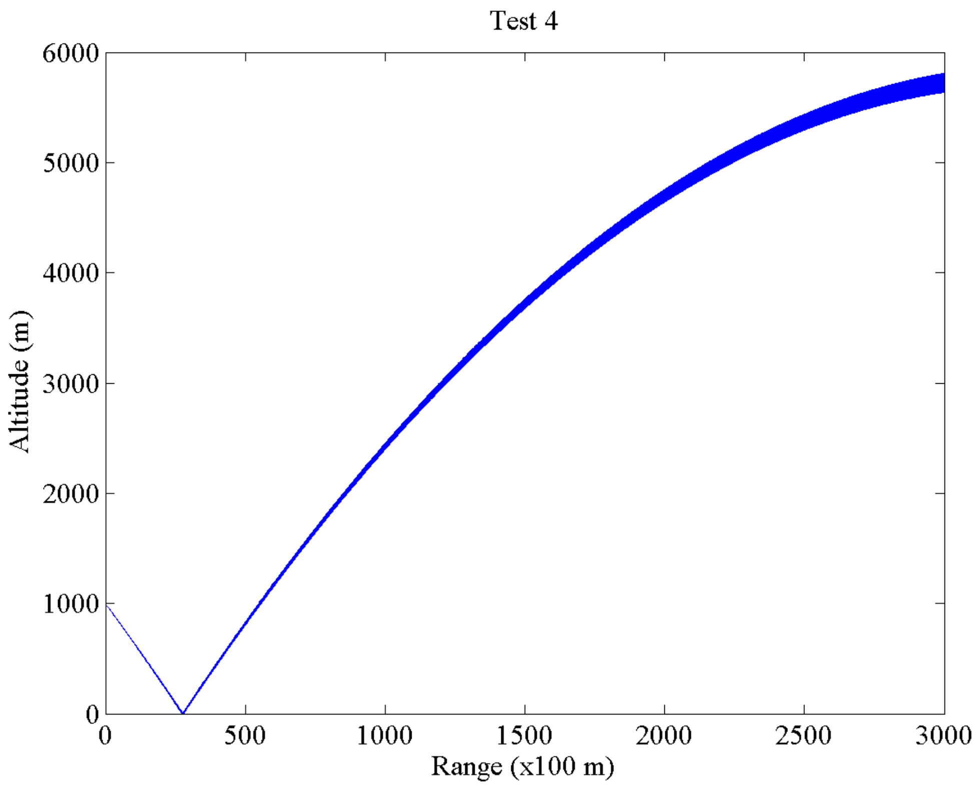


Figure 5.7: Map2 & RaySet1

Table 5.15: RWPRayMap Times of Test-2-1

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
25.81	7.54	17.23	2.25	30.22	31.96	43.28	45.30

Table 5.16: RWPRayTarget Times of Test-2-1

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
27.02	7.48	17.51	2.39	33.83	33.86	45.14	45.18

and test-1-1 show similarities. In both tests both CPUs are faster than both GPUs and GPU1 outperforms GPU2. Moreover, supremacy of CPUs is clearer than test-1-1. Just to remember, this test uses the same ray set with test-1-1 and a denser map than it.

Performance sorting for test-2-1:

CPU2p > CPU1p > CPU2s > CPU1s > GPU1 > GPU2

Table 5.17: RWPRayMap Speedup for Test-2-1

	GPU1		GPU2	
	Serial	Parallel	Serial	Parallel
CPU1	0.85	0.25	0.60	0.17
CPU2	0.57	0.08	0.40	0.05

All speedup values are less than 1 and GPU1 has the greater speedups.

5.4.5 Test-2-2

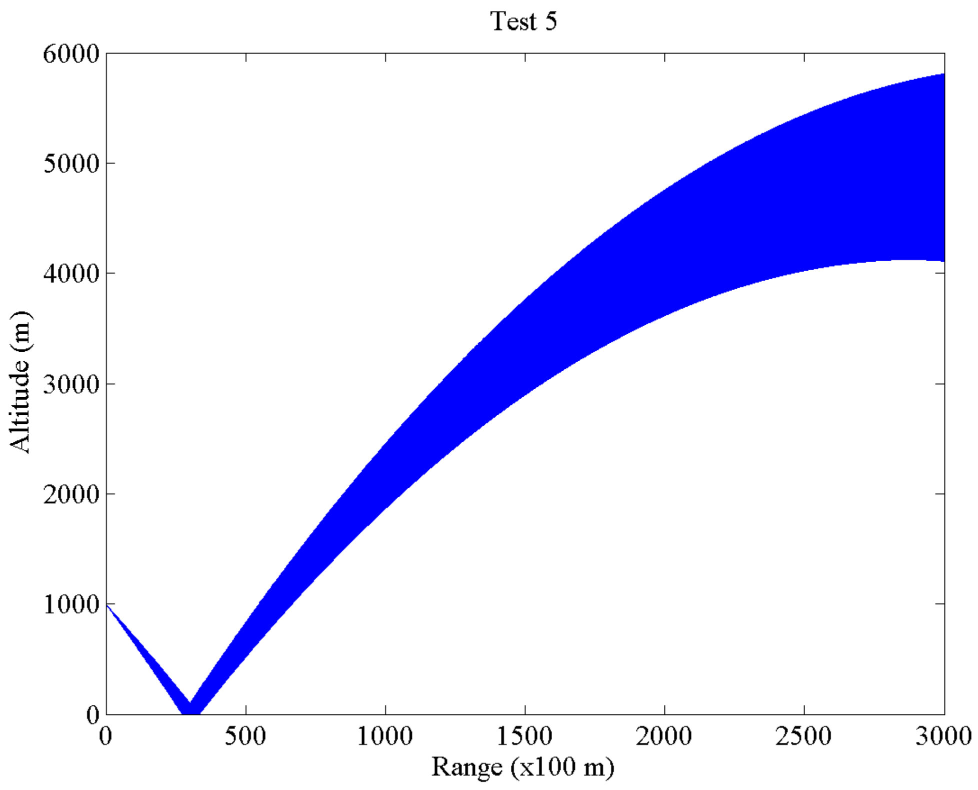


Figure 5.8: Map2 & RaySet2

Figure 5.8 shows resultant radio wave propagation map of test-2-2. In this test, rays are spanning an area with 0.4° on map2. This test is run for the same field with test-1-2, with a higher resolution. Resultant view is already the same with the one of test-1-2 but output of this test consist of 10 times larger data.

In Table 5.18 and Table 5.19 running time measurements of RWPRayMap and RWP-RayTarget are shown respectively. It is easy to see the similarities between test-2-2

Table 5.18: RWPRayMap Times of Test-2-2

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
261.24	69.85	176.84	16.88	46.85	66.04	50.26	67.96

Table 5.19: RWPRayTarget Times of Test-2-2

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
264.05	72.14	178.33	18.03	50.28	50.33	51.50	51.55

and test-1-2 results, which use the same ray set. In both tests, GPUs are faster than serial implementation of CPUs a few times. GPUs are also faster than parallel implementation of CPU1 but not CPU2. Moreover, GPU1 is better than GPU2 according to test-2-2. This ordering is just the same with ordering of test-1-2.

Performance sorting for test 5:
CPU2p > GPU1 > GPU2 > CPU1p > CPU2s > CPU1s

Table 5.20: RWPRayMap Speedup for Test-2-2

	GPU1		GPU2	
	Serial	Parallel	Serial	Parallel
CPU1	5.58	1.49	5.20	1.39
CPU2	3.77	0.36	3.52	0.34

Only the parallel implementation running on CPU2 has a speedup less than 1 and GPU1 has the greater speedups. Moreover, the highest speedup is 5.58.

5.4.6 Test-2-3

Propagation map of test-2-3 is shown in Figure 5.9. In this test, rays are spanning an area of 4°. Because this map is too crowded, which is a property of ray set 3, and it is hard to identify the rays, we also show a reduced version of it in Figure 5.10. In that reduced figure, instead of all rays, every 100^{th} ray is plotted, just like test-1-3 results. View of the results are just the same with test-1-3, with a higher resolution, which means 10 times more output data.

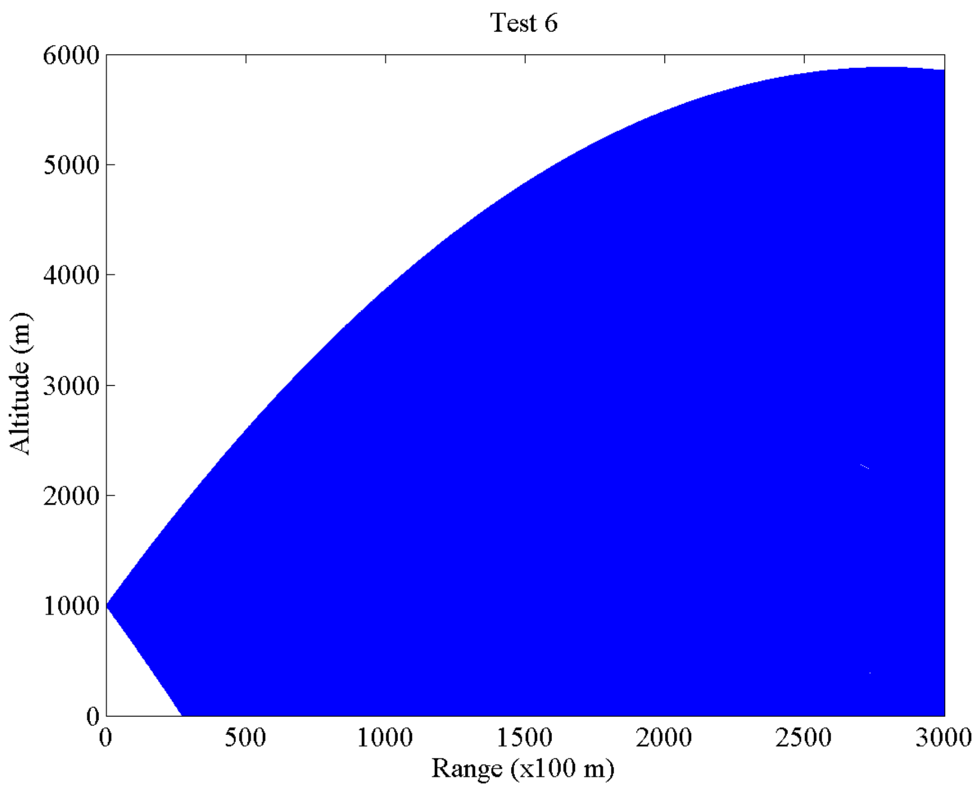


Figure 5.9: Map2 & RaySet3

Table 5.21: RWPRayMap Times of Test-2-3

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
2595.75	694.15	1744.82	162.01	167.62	333.94	103.51	276.44

Table 5.22: RWPRayTarget Times of Test-2-3

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
2584.24	687.26	1742.74	171.85	161.93	162.11	105.59	105.79

Running time results of RWPRayMap and RWPRayTarget are shown in Table 5.21 and Table 5.22 respectively. According to this results, both GPUs are working better than both CPUs. Better GPU is nearly 17 times faster than better sequential CPU. Both GPUs are even better than both parallel CPUs. These orderings are just like test-1-3's ones. Even, GPU2 is giving better performance than GPU1. More crowded ray sets profit GPUs over CPUs and GPU2 over GPU1.

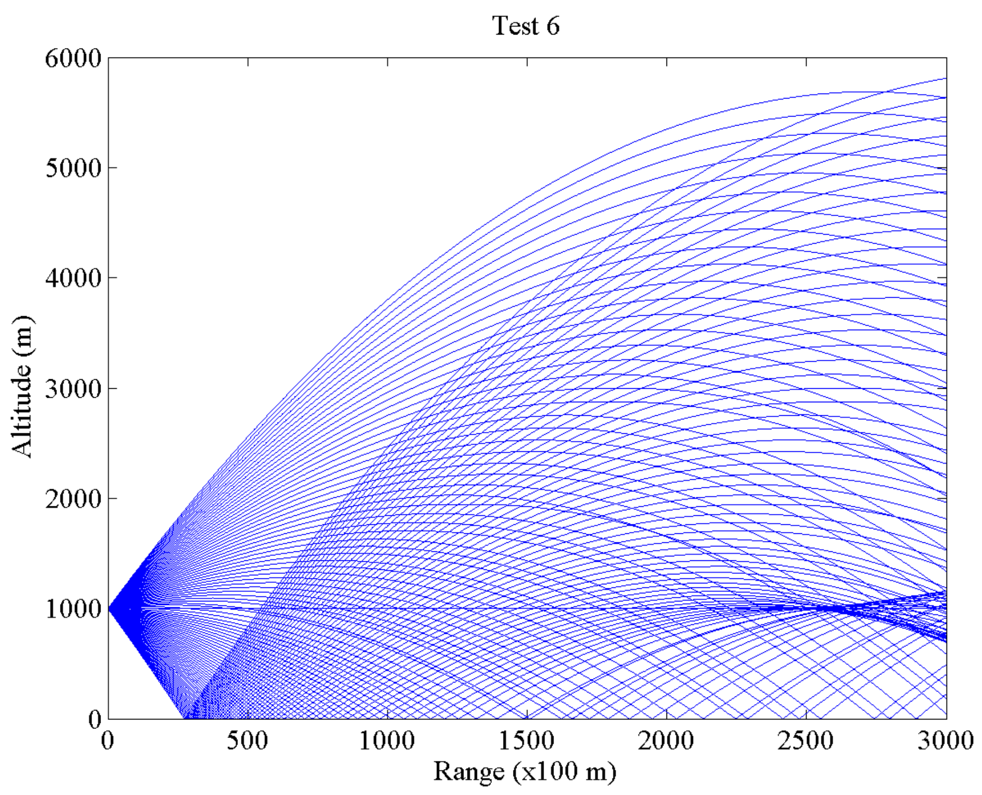


Figure 5.10: Map2 & RaySet3 (Reduced)

Performance sorting for test-2-3:

GPU2 > CPU2p > GPU1 > CPU1p > CPU2s > CPU1s

Table 5.23: RWPRayMap Speedup for Test-2-3

	GPU1		GPU2	
	Serial	Parallel	Serial	Parallel
CPU1	15.49	4.14	25.08	6.71
CPU2	10.41	0.97	16.86	1.57

All speedup values are greater than or near to 1. GPU2 has the greater speedups. Moreover 25.08 is the highest speedup.

5.4.7 Test-3-1

Figure 5.11 shows radio wave propagation map of test-3-1, which uses map3 and ray set 1. Rays are spanning an area of 0.04° in this test. This map has a grater range than the previous ones. Hence we can fallow the rays until a further field. Rays are

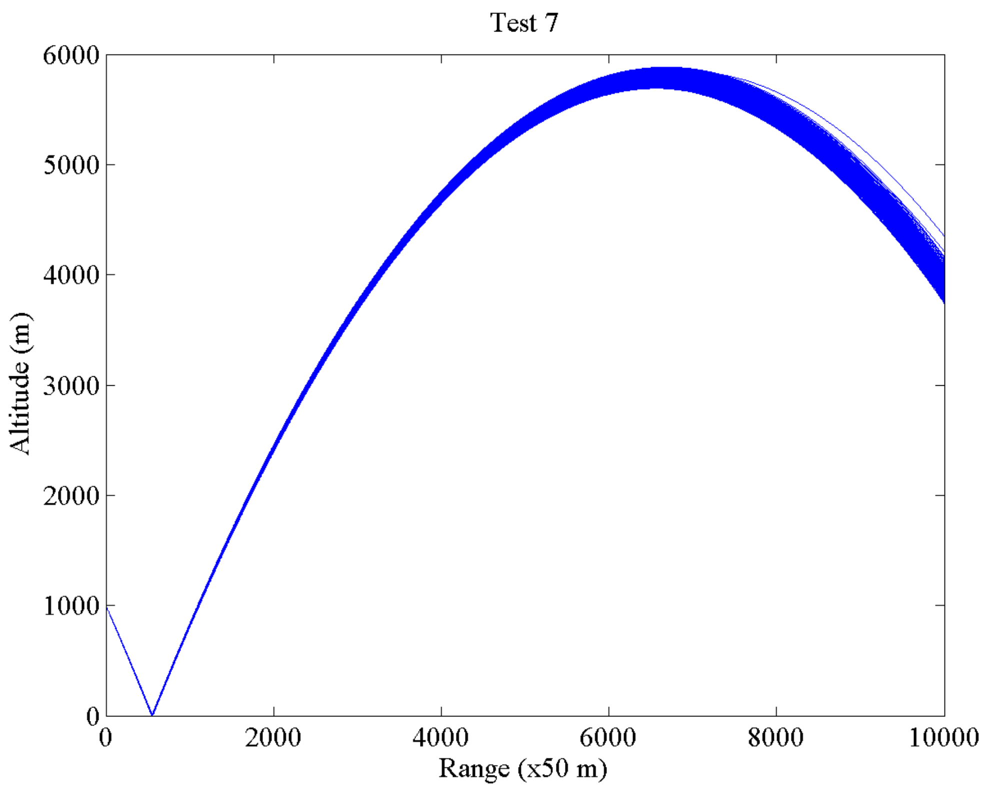


Figure 5.11: Map3 & RaySet1

reflecting from terrain, going upward, and then, as a result of refraction, they turn again to a downward direction. This is seen in this test in favor of high range of the map. Rays are spanning a wider area while they go further the transmitter point.

Table 5.24: RWPRayMap Times of Test-3-1

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
85.35	24.62	57.45	6.36	105.13	111.30	145.20	151.06

Table 5.25: RWPRayTarget Times of Test-3-1

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
84.56	24.02	58.20	6.91	114.11	114.13	151.20	151.24

Table 5.24 shows running time results of RWPRayMap and Table 5.25 shows running time results of RWPRayTarget for test-3-1. From these time values, we clearly see that both CPUs are working better than both GPUs for this test. We actually got used

to see such a result for the tests that are using ray set 1. Thus we did not got surprised from these tables. The other thing that we got used to see is that, GPU1 is working better from GPU2 for this ray set.

Performance sorting for test-3-1:
CPU2p > CPU1p > CPU2s > CPU1s > GPU1 > GPU2

Table 5.26: RWPRayMap Speedup for Test-3-1

	GPU1		GPU2	
	Serial	Parallel	Serial	Parallel
CPU1	0.81	0.23	0.59	0.17
CPU2	0.55	0.06	0.40	0.04

All speedup values are less than 1 and GPU1 has the greater speedups.

5.4.8 Test-3-2

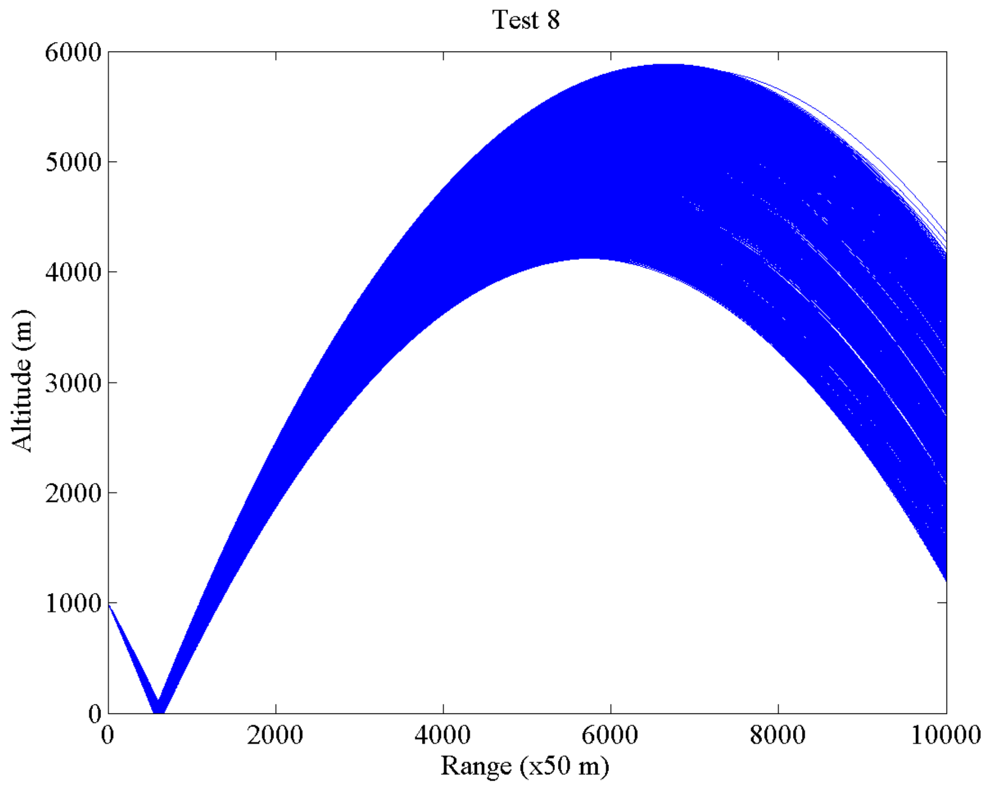


Figure 5.12: Map3 & RaySet2

Radio wave propagation map of test-3-2 is shown in Figure 5.12. Ray set 2 is used in this test and rays are spanning an area of 0.4°. Rays are reflecting from the terrain, however none of them is leaving the map from the top. They are coming back to a downward direction with the effect of refraction.

Table 5.27: RWPRayMap Times of Test-3-2

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
852.13	235.91	577.29	55.48	162.64	219.95	168.86	232.33

Table 5.28: RWPRayTarget Times of Test-3-2

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
849.37	235.34	580.56	57.97	175.03	175.07	172.64	172.70

Running time results of RWPRayMap are shown in Table 5.27 and running time results of RWPRayTarget are shown in Table 5.28. According to mentioned running time measurements, both GPUs are giving better results from both CPUs. Note that parallel run on CPU2 is completing test-3-2 first. Moreover, memory operation times for GPUs are significant, and this situation gives the advantage to RWPRayTarget against RWPRayMap on GPUs.

Performance sorting for test-3-2:
CPU2p > GPU1 > GPU2 > CPU1p > CPU2s > CPU1s

Table 5.29: RWPRayMap Speedup for Test-3-2

	GPU1		GPU2	
	Serial	Parallel	Serial	Parallel
CPU1	5.24	1.45	5.05	1.40
CPU2	3.55	0.34	3.42	0.33

Only the parallel implementation running on CPU2 has a speedup less than 1 and GPU1 has the greater speedups. Moreover, the highest speedup is 5.24.

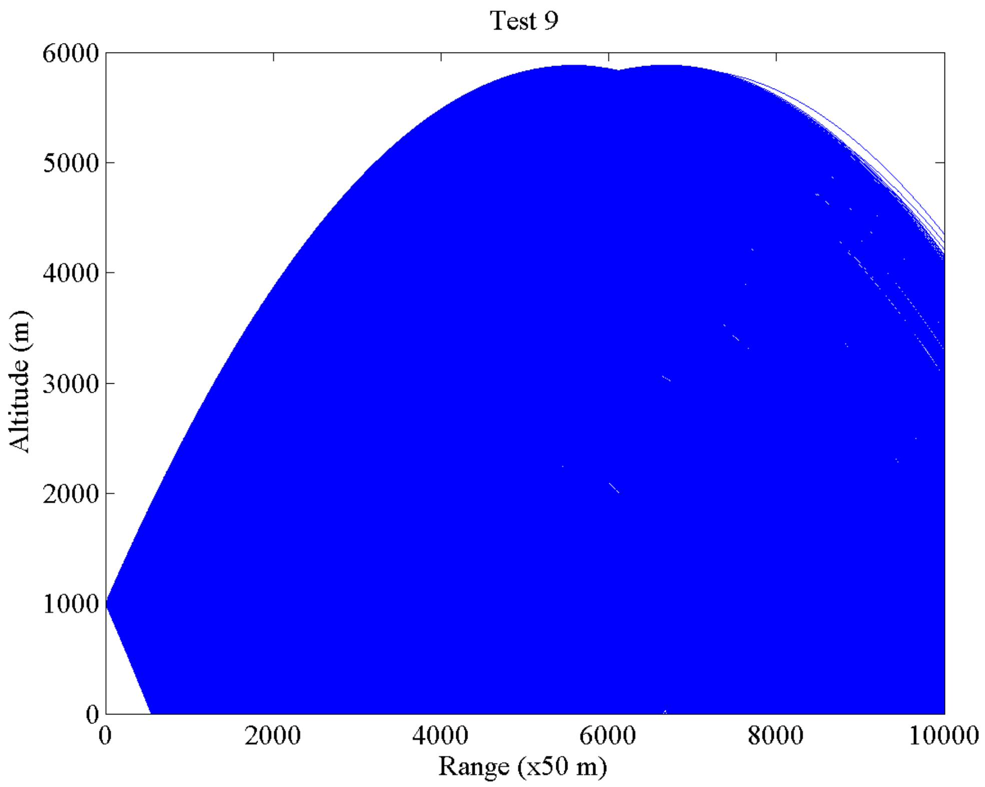


Figure 5.13: Map3 & RaySet3

5.4.9 Test-3-3

Figure 5.13 shows propagation map of test-3-3. Figure 5.14 shows a reduced version of radio wave propagation map of test-3-3. In that reduced figure, instead of all rays, every 100^{th} ray is plotted, just like test-1-3 and test-2-3 results. Rays are spanning an area of 4° in this test. Effects of reflection and refraction can be seen from that figures.

Table 5.30: RWPRayMap Times of Test-3-3

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
8526.7	2306.9	5949.2	535.7	558.4	1127.7	351.2	930.1

Table 5.31: RWPRayTarget Times of Test-3-3

CPU1		CPU2		GPU1		GPU2	
Serial	Parallel	Serial	Parallel	Pure	Full	Pure	Full
8530.6	2343.3	5827.0	571.8	544.8	545.0	357.8	358.0

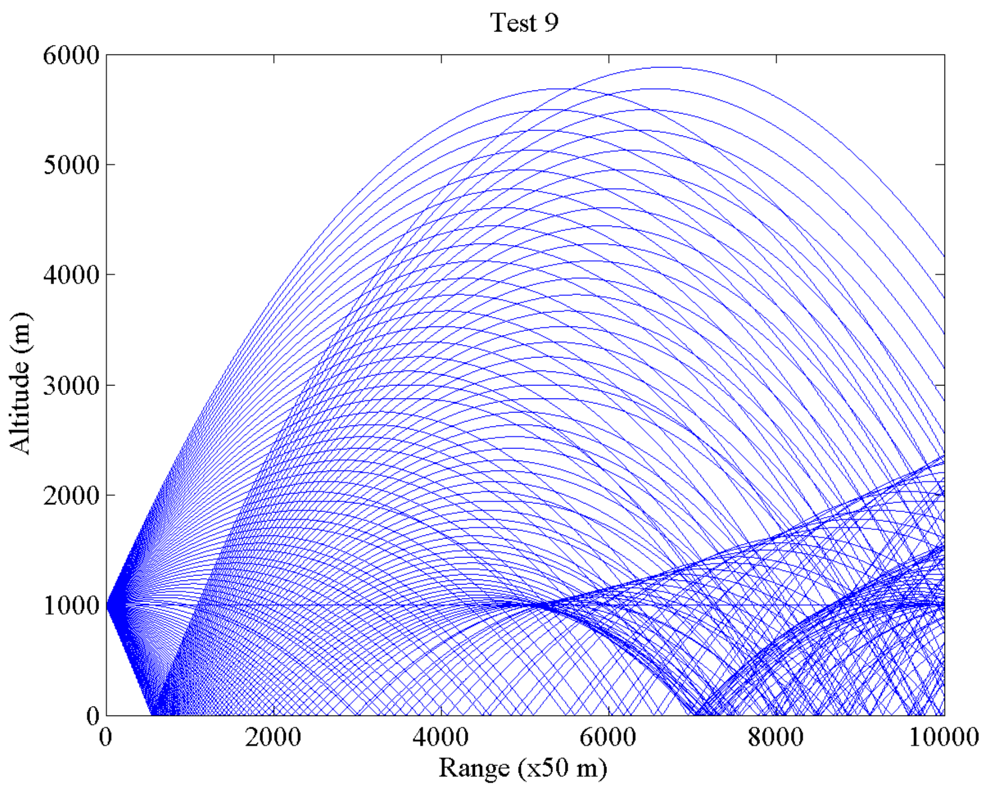


Figure 5.14: Map3 & RaySet3 (Reduced)

Running time measurements of RWPRayMap and RWPRayTarget are shown in Table 5.30 and Table 5.31 respectively. As a classic of ray set 3, both GPUs give better performance than both CPUs. Moreover, faster GPU gives the result before the faster CPU even on parallel implementation. GPU2 becomes the better GPU instead of GPU1 for ray set 3. Test-3-3 has the greatest input and output data, therefore, memory operation times, which are greater than process times, are becoming more and more significant, which makes RWPRayTarget preferable to RWPRayMap.

Performance sorting for test-3-3:
GPU2 > CPU2p > GPU1 > CPU1p > CPU2s > CPU1s

Table 5.32: RWPRayMap Speedup for Test-3-3

	GPU1		GPU2	
	Serial	Parallel	Serial	Parallel
CPU1	15.27	4.49	24.28	6.57
CPU2	10.65	0.96	16.94	1.53

All speedup values are greater than or near to 1. GPU2 has the greater speedups.

Moreover 24.28 is the highest speedup.

5.4.10 Comperative Analysis of Results

Table 5.33: RWPRayMap Times of all Tests

	CPU1		CPU2		GPU1		GPU2	
Test #	Serial	Par.	Serial	Par.	Pure	Full	Pure	Full
Test-1-1	2.887	1.719	2.399	**1.475**	3.166	3.495	4.403	4.707
Test-1-2	26.674	7.623	19.280	**2.781**	4.732	6.857	5.066	6.920
Test-1-3	258.80	70.48	188.58	18.43	15.75	34.97	**10.39**	27.70
Test-2-1	25.81	7.54	17.23	**2.25**	30.22	31.96	43.28	45.30
Test-2-2	261.24	69.85	176.84	**16.88**	46.85	46.87	50.26	67.96
Test-2-3	2595.75	694.15	1744.82	162.01	167.62	333.94	**103.51**	276.44
Test-3-1	85.35	24.62	57.45	**6.36**	105.13	111.30	145.20	151.06
Test-3-2	852.13	235.91	577.29	**55.48**	162.64	219.95	168.86	232.33
Test-3-3	8526.7	2306.9	5949.2	535.7	558.4	1127.7	**351.2**	930.1

Running time measurements of RWPRayMap for all tests are collected in Table 5.33 to be able to see them all together. Shortest time of each test is marked with **bold**. According to this table, CPU1 and GPU1 could not win any of the tests. Actually in the tests that do not use ray set 3, GPU1 could beat GPU2, however it fell behind CPU2 parallel version.

Table 5.34: RWPRayMap Serial Speedup (CPU2 - GPU1)

	Ray Set 1	Ray Set 2	Ray Set 3
Map 1	0.76	4.07	11.97
Map 2	0.57	3.77	10.41
Map 3	0.55	3.55	10.65

Table 5.35: RWPRayMap Serial Speedup (CPU2 - GPU2)

	Ray Set 1	Ray Set 2	Ray Set 3
Map 1	0.54	3.81	18.14
Map 2	0.40	3.52	16.86
Map 3	0.40	3.42	16.94

Table 5.34 shows the speedup values between CPU2 and GPU1, and Table 5.35 shows the speedup values between CPU2 and GPU2. Speedup values in comparison with

CPU1, which are all higher from these, are not collected here, because CPU2 has given a better performance for all the tests. According to these speedup tables, independently of the maps, the more rays we have, the more GPU speedup we get.

As mentioned in Section 2.3, GPUs have a great number of cores and a higher latency than CPUs. To be able to take advantage of these all cores and close the latency deficit, we need to split the work into a lot of, small, identical parts. In ray tracing, we accept each ray as the unit of work and do not try to split the work of a single ray. Actually it is already not easy to do that because calculation of a single ray is exactly sequential. Each step is directly dependent to the previous one. Thus, calculating one ray, is accepted to be impartible and must be calculated by one single thread. Thus, to be able to get high performance from GPUs, we need to increase the ray count, which are workpieces, and decrease the ray size (map size), which is size of the each workpiece. Moreover, we need to occupy all cores of the GPU by giving it a huge number of small workpieces to distribute to the cores. Just to remember, as we mentioned in Table 5.2 GPU1 has 448 cores and GPU2 has 2496 cores. Moreover, ray set 1 has 100 rays, which are quite less than the core counts. When the total work is so small, the latency come into prominence and do not let the GPU to give a gain in performance. This clearly explains why speedup values of tests that use ray set 1 are less than 1.

Table 5.36: RWPRayMap Winner GPU

	Ray Set 1	Ray Set 2	Ray Set 3
Map 1	GPU1	GPU1	GPU2
Map 2	GPU1	GPU1	GPU2
Map 3	GPU1	GPU1	GPU2

Ray set 3 is consist of 10000 rays. When we compare core numbers of the GPUs, it is not surprising to see that GPU2 has better performance on ray set 3. Moreover, GPU2 has a higher memory speed and higher double precision performance. Table 5.36 shows the winner GPU in these tests. The used map can not change the champion, however the ray set assigns it directly. Small number of rays are not giving good results on GPU2 because of its frequency of CUDA cores. As we mentioned in Table 5.2 clock speed of GPU1 is 1.15 GHz and clock speed of GPU2 is 706 MHz. Because ray sets 1 and 2 are too small for GPU2, GPU1 is beating it for these tests under favor of its clock speed. When the ray set gets larger, favor of core number overcomes favor of clock speed.

While parallelizing a work to implement it for GPU, the work must be partitioned to a great number of small, identical and independent work pieces. Let us name each of the work pieces as a unit work. To get the best performance from GPU, size of a unit work should be minimized, and count of the unit works should be maximized. Calculation of a unit work, should be independent from the others, and if it is possible, independent from the memory. Memory operations decrease the performance and cause to higher computing durations. High usage level of memory is one of the most common reasons that cause to low GPU performance. Decreasing the memory operations will come back with an increase of the performance. When it is not possible to avoid from memory usage, at least it should be done regularly. It is more efficient to read consecutive parts of the memory by consecutive CUDA threads. When memory usage of the threads is inevitable, the programmer may still increase the performance, organizing memory need of each thread in an efficient structure.

In our problem, during the ray tracing calculations, refractive index of each position is read from the memory, for each step of the ray, which is too often for a GPU implementation. Moreover, the rays get further away from each other and this situation breaks the regularity of the memory usage. Each thread reads another irrelevant part of the memory. This situation is a reason that decreases the performance of our GPU implementation. It is possible to get a much better performance from the GPU with a version that does not calculate the refraction, or calculates it with a constant refractive indices ratio value to get rid of the memory necessity. However, the refraction and its results, is one of the main purposes of this study.

Another point we may touch out is the parallelization speedup of the CPUs. Table 5.37 shows parallelization speedup values of CPU1 and CPU2. Just to remember, CPU1 is one processor with 4 cores, and CPU2 is two processors with 8 cores each. Thus, ideal speedup of CPU1 is 4 and ideal speedup of CPU2 is 16.

According to Table 5.37, speedup of CPU1 changes between 3.42 and 3.74. We offer to ignore test-1-1 because of its simplicity. These speedup values are very close to ideal speedup. The lowest one is 86% and the highest one is 94% of the ideal, which

Table 5.37: Parallelization Speedup of CPUs

	CPU1	CPU2
Test-1-1	1.68	1.63
Test-1-2	3.50	6.93
Test-1-3	3.67	10.23
Test-2-1	3.42	7.66
Test-2-2	3.74	10.48
Test-2-3	3.74	10.77
Test-3-1	3.47	9.03
Test-3-2	3.61	10.41
Test-3-3	3.70	11.11

is theoretically maximum speedup.

However, speedup values of CPU2 are not so close to the ideal one. They change between 6.93 and 11.11. Even the best value 11.11 is about only 70% and the lowest one is 43% of the ideal speedup. There may be several reasons of this difference. The first one coming to mind is that CPU1 is a single processor and CPU2 is collaboration of two processors. Because the communication between the processors is slower than the communication between the cores, CPU2 is starting to parallelization race one step behind. The other important point is that, running 16 threads on both 4 core system and 16 core system is not fair. Running only one thread per core may cause idle times in the cores, which negatively affects the parallelization speedup. When we run 16 threads on 4 cores, it may be better to run 64 threads on 16 cores. However, we do not focus on such measurements in this thesis because this is out of scope of this study.

CHAPTER 6

CONCLUSION

In this thesis, we introduced a new radio wave propagation modeling tool which is called RWPRay. In this study, we used ray tracing technique for the modeling. Reflection, refraction and free space path loss are implemented within the scope of this study. Five different predefined surface types and five predefined atmosphere types are included in RWPRay. Moreover, the user may define specific surface types and specific atmosphere types.

RWPRay has two main functionalities. The first functionality is that it is able to create radio wave propagation map of a field according to the transmitter and environmental parameters. The other functionality is that it is able to find the rays that hit or pass close to a specified target in a field, according to the same parameters.

RWPRay is implemented for CPU and GPU domains separately. It has two versions developed for CPU which are running in sequential and parallel manners. In this thesis, performances of all three versions of RWPRay are presented and analyzed.

The aim of this study is to accelerate radio wave propagation modeling with GPU computing. This acceleration makes a major contribution to radar environment simulators, which are needed by radar producers. Radar environment simulators save budget, time and labor force of radar developers. Moreover, contributing the radar producers means contributing national defense industry.

As a result of this study, we achieved up to 18.14 speedup values between recent, high specification CPU and GPU cards. GPU computing performance gain may differ from application to application, because not all problems and solutions are favorable for architecture and structure of GPU. As a results, we should not expect the same speedup values from different applications.

As a future work, first extension for RWPRay may be implementing the scattering and diffraction properties of radio waves. When an electromagnetic wave hits a surface, beside reflection, it also separates and radiates in different directions according to smoothness of the surface, which is called scattering. Scattering breaks the parallelization layout, which makes it difficult to calculate in GPU computing manner. Diffraction is a phenomena that causes the ray to reflect from sharp edges with a different angle. It may induce radio waves to be received from points that are not in line of sight area.

Secondly, propagation factor of waves may be also calculated [13]. In RWPRay total range and current power is calculated for each step of the rays, but propagation factor is not calculated.

Finally, the real 3D propagation may be implemented. RWPRay splits the 3D map to 2D slices and ignores the interaction in between of them. For a more accurate 3D modeling, rays move on all x, y and z directions and effects like reflection, scattering and diffraction expand to 3D.

These possible extensions enhance the accuracy of radio wave propagation modeling. However, because we are interested in GPU acceleration and CPU-GPU performance comparison more than the propagation modeling itself, we left these as future studies.

REFERENCES

[1] J. Agnarsson. Simulation of a Radar In Flames : A Ray Based Radar Model. Technical Report 13012, Uppsala University, Department of Information Technology, 2013.

[2] T. E. Athanaileas, G. E. Athanasiadou, G. V. Tsoulos, and D. I. Kaklamani. Parallel Radio-wave Propagation Modeling with Image-based Ray Tracing Techniques. *Parallel Computing*, 36(12):679–695, December 2010.

[3] L. Barclay. *Propagation of Radiowaves*. IET electromagnetic waves series. Institution of Engineering and Technology, 2013.

[4] A. M. Cavalcante, M. J. Sousa, J. C. W. A. Costa, C. R. L. France, G. P. S. Cavalcante, and Jr C. S. Sales. 3D Ray-Tracing Parallel Model for Radio-Propagation Prediction. *VI International Telecommunications Symposium*, September 2006.

[5] B. R. Epstein and D. L. Rhodes. GPU-accelerated ray tracing for electromagnetic propagation analysis. In *IEEE International Conference onWireless Information Technology and Systems (ICWITS)*, pages 1–4, August 2010.

[6] A. Ghasemi, A. Abedi, and F. Ghasemi. *Propagation Engineering in Wireless Communications*. Springer, 2011.

[7] Khronos Group. reflect - OpenGL 4 Reference Pages, 2013. `https://www.opengl.org/sdk/docs/man/html/reflect.xhtml` [Online; accessed 17-July-2014].

[8] Khronos Group. refract - OpenGL 4 Reference Pages, 2013. `https://www.opengl.org/sdk/docs/man/html/refract.xhtml` [Online; accessed 17-July-2014].

[9] C. P. Hattan. A Ray Trace Model for Propagation Loss. Technical Report ADA266314, Naval Command Control and Ocean Surveillance Center Rdt and E div, San Diego, CA, February 1993.

[10] Nvidia Inc. NVIDIA Fermi Compute Architecture Whitepaper, December 2009. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[11] Nvidia Inc. NVIDIA Kepler GK110 Architecture Whitepaper, May 2012. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[12] S. S. John. *Introduction to RF Propagation*. Wiley, 2005.

[13] H. T. Meng. Acceleration of Asymptotic Computational Electromagnetics Physical Optics - Shooting and Bouncing Ray (PO-SBR) Method Using CUDA. Master's thesis, University of Illinois, 2011.

[14] O. Ozgun, G. Apaydin, M. Kuzuoglu, and L. Sevgi. PETOOL: MATLAB-based one-way and two-way split-step parabolic equation tool for radiowave propagation over variable terrain. *Computer Physics Communications*, 182(12):2638–2654, 2011.

[15] J. A. Richards. *Radio Wave Propagation: An Introduction for the Non-Specialist*. Springer, 2008.

[16] S. Y. Seidel and T. S. Rappaport. Site-specific propagation prediction for wireless in-building personal communication system design. *IEEE Transactions on Vehicular Technology*, 43(4):879–891, November 1994.

[17] L. Sevgi. A Ray-Shooting Visualization MATLAB Package for 2D Ground-Wave Propagation Simulations. *Antennas and Propagation Magazine, IEEE*, 46(4):140–145, August 2004.

[18] P. Shirley and S. Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 3rd edition, 2009.

[19] B. Sklar. *Digital communications: fundamentals and applications*. Prentice Hall Communications Engineering and Emerging Technologies Series. Prentice-Hall PTR, 2001.

[20] M. I. Skolnik. *Introduction to Radar Systems*. McGraw-Hill, 1962.

[21] M. I. Skolnik. *Radar Handbook, Third Edition*. Electronics electrical engineering. McGraw-Hill Education, 2008.

[22] F. Ticconi, L. Pulvirenti, and N. Pierdicca. *Models for Scattering from Rough Surfaces, Electromagnetic Waves*. InTech, June 2011.

[23] Wikipedia. Ray tracing (graphics) — Wikipedia, The Free Encyclopedia, 2014. http://en.wikipedia.org/w/index.php?title=Ray_tracing_(graphics) [Online; accessed 15-July-2014].

[24] Wikipedia. Ray tracing (physics) — Wikipedia, The Free Encyclopedia, 2014. http://en.wikipedia.org/w/index.php?title=Ray_tracing_(physics) [Online; accessed 17-July-2014].

APPENDIX A

MAP CREATOR

We have developed a tool called Map Creator as a helper program for RWPRay. Map Creator generates terrain and surface type maps in the format that RWPRay accepts. It creates both binary and ASCII files of the maps. RWPRay reads the binary file, and the ASCII file is for human read. Both height map and surface type map are written to this files, just one after the other.

Map Creator accepts following parameters:

mapType : 2D or 3D

mapSize : Size of the map

mapWidth : Width of the map for 3D maps

mapResolution : Resolution of the map, in meters

heightLimit : The maximum altitude value that map can show, in meters

Apart from these parameters, user may define up to 10 points, giving their range and altitude values. User may also define surface types of these regions. Map Creator creates the map, interpolating medium points linearly according to user defined points. Map Creator and RWPRay use 32 bit float for map height values.