Large-Scale Motion Modelling using a Graphical Processing Unit

LARGE-SCALE MOTION MODELLING USING A GRAPHICAL PROCESSING UNIT

BY

YUSUF DINATH, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2012)	McMaster University
(Electrical & Computer Engineering)	Hamilton, Ontario, Canada

TITLE:	Large-Scale Motion Modelling using a Graphical Processing Unit
AUTHOR:	Yusuf Dinath B.Eng., (Electrical Engineering) McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. T. Kirubarajan
NUMBER OF PAGES:	xiii, 59

Abstract

The increased availability of Graphical Processing Units (GPUs) in personal computers has made parallel programming worthwhile and more accessible, but not necessarily easier. This thesis will take advantage of the power of a GPU, in conjunction with the Central Processing Unit (CPU), in order to simulate target trajectories for large-scale scenarios, such as wide-area maritime or ground surveillance. The idea is to simulate the motion of tens of thousands of targets using a GPU by formulating an optimization problem that maximizes the throughput. To do this, the proposed algorithm is provided with input data that describes how the targets are expected to behave, path information (e.g., roadmaps, shipping lanes), and available computational resources. Then, it is possible to break down the algorithm into parts that are done in the CPU versus those sent to the GPU. The ultimate goal is to compare processing times of the algorithm with a GPU in conjunction with a CPU to those of the standard algorithms running on the CPU alone. In this thesis, the optimization formulation for utilizing the GPU, simulation results on scenarios with a large number of targets and conclusions are provided.

Acknowledgements

First and foremost I would like that thank my supervisor, Dr. T. Kiruba, for allowing me to be his research student during my undergrad studies, a summer student after completing my undergrad degress and finally his master's degree student for the past two years. Also, I would like to thank Dr. R. Tharmarasa for his endless support, guidance and direction.

I am grateful for the opportunity that Dr. M. McDonald extended during my first summer as a master's student, which allowed me to work in Ottawa for the Defence Research and Development Canada in the Radar Systems group. It provided me with valuable knowledge and work experience. I also thank Dr. A. Patriciu for sharing his knowledge and understanding of how to program in parallel using the GPU.

I appreciate and would like to thank Dr. I. Bruce and Dr. A. Jeremic for being my committee members and for providing feedback on my thesis.

Finally, I would also like to thank Mrs. C. Gies, Graduate Administrative Assistant, who ensures that the ECE department runs smoothly.

Abbreviations

1D	One-Dimensional Space
2D	Two-Dimensional Space
3D	Three-Dimensional Space
BIH	Bureau International de l'Heure
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
ECEF	Earth-Centered, Earth-Fixed
GFLOPS	Giga Floating-Point Operations Per Second
GPU	Graphical Processing Unit
GPGPU	General Purpose Graphical Processing Unit
ICAO	International Civil Aviation Organization
LSMM	Large-Scale Motion Models
PC	Personal Computer
WGS-84	World Geodetic System - 1984

Notation

Γ	Total noise added to a state
λ	Longitude
ν	White noise vector
ω	Turn rate
ϕ	Latitude
θ	Target's heading
υ^{λ}	Ground velocity, longitudial-wise
υ^{ϕ}	Ground velocity, latitude-wise
a	Semi-major axis
b	Semi-minor axis
e	First eccentricity
f	Ellipsoidal flattening (or oblateness)
F	State transition natrix
G	Gain matrix
GB/s	Gigabytes per second
h	Altitude
L	Number of targets
k	Time step

m	Metre
M_L	Number of motion legs for a target
M_m	Number of Monte Carlo runs
R^m	Radii of curvature in the meridian
R^v	Radii of curvature in the prime vertical
S	Speed up factor of two execution times
s	Second
T	Time difference between k and $(k-1)$
$t(\cdot)$	Execution time of a program
x_k	State at time k

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Problem Statement

The goal of this thesis is to develop a method to simulate large-scale multitarget scenarios, which takes advantage of the parallelization power provided by the Graphical Processing Unit (GPU) in combination with the Central Processing Unit (CPU). With this method and simulations, it will be shown how the combined use of GPU and CPU to simulate scenarios is an improvement compared to using the CPU alone. To simulate scenarios, the algorithm for the GPU will be broken down into a multistage process, showing exactly how to parallelize the CPU's sequential process of generating scenarios and showing the math behind the parallelized code.

The CPU's performance has consistently improved, according to Moore's Law, based on clock frequency and the number of transistors. However, the CPU does not only execute one algorithm, it also carries out instructions for the entire system. GPU were initially designed for rapid processing for graphical output. With GPUs becoming widely available, with dramatic improvements to computational speed and with the use of GPUs in many different fields, it is now considered a General Purpose Graphic Processing Unit (GPGPU). The GPU is very efficient at performing tasks in parallel, both graphical and non-graphical, and many researchers and organizations have realized the GPU's potential (J.D. Owens, 2005). It must be noted that even though the performance of GPUs and CPUs have significantly improved, the performance pales in comparison to the improvements made to programming algorithms optimizations (J.P. Holdren and Varmus, 2010). New programming approaches are needed to take full advantage of the parallel processing power.

Simulating large-scale multitarget scenarios is important when trying to test the performance of trackers. Tracking is the process of estimating a target's current and future state based on the target's prior information and past and current measurements. The target trajectories that are generated from the simulation can be used as baseline tests for trackers, and they must be simple to create and modify. Using motion models to create target trajectories is important because the motion models allow exact control over how the targets move and behave, thus having the ability to test trackers in various scenarios quickly. Creating target trajectories is the first step in a multistage process; it is important that this process is a quick and simple.

With advanced tracking algorithms, it is no longer good enough simply to have a handful of targets to check the performance of trackers; large-scale scenarios are a must. Thousands of targets could occupy a relatively small area. For example, pedestrians in a downtown core, vehicles on a busy road, watercraft near a port or aircraft near an airport. Targets can also be either path constrained or unconstrained. Path constrained targets have to follow a predefined path. A path is made up of segments and waypoints, and paths can come to intersections (branch, merge or cross) (T. Kirubarajan and Kadar, 2000). A path can be defined by one of many motion models, and can have various restrictions, such as speed, acceleration, width, etc. Constrained targets deviating from a path is a highly unlikely event, whereas unconstrained targets can freely move in any direction without any restrictions.

For targets such as pedestrians, vehicles and short range watercraft, it is possible to run simulations in a two-dimensional (2D) environment. A local coordinate system is acceptable when using short-range targets. However, for long-range vessels such as large watercraft or any aircraft, the curvature of the earth must be taken into account. Therefore, the simulations must be generated in a three-dimensional (3D) environment. The earth is best represented as a specialized ellipsoid, where the distance from the North Pole to South Pole is shorter than the distance east to west. To represent the earth as an ellipsoid, the World Geodetic System - 1984 (WGS-84) standard is used. Using this standard, it is easy to plot targets trajectories in Cartesian coordinates, or even have the target trajectories in geographic coordinates (longitude, latitude and altitude). (ICAO, 2002)

The main contributions provided by this thesis are as follows:

- Parallelizing the target generation by parallelizing the motion legs:

  There are multiple ways to generate target trajectories. The way presented in this thesis is done by parallelizing a target's motion legs; every element in the motion leg is calculated at once.

- A new general state equation must be derived:

  The state equation must be manipulated such that it can be run in parallel. Adding noise to the non-Cartesian elements makes parallelizing the state equation difficult.

- The state vector $\mathbf{x_k}$ is broken up into two separate parts:

  One part of the state equation is for Cartesian elements and the other part is for non-Cartesian elements (such as the turn rate, ω). It is assumed that the state transition matrix is only dependent on the non-Cartesian elements.

The remainder of the thesis is structured as follows. Chapter 2 provides background information on other targets, coordinate systems, process noise and motion models (in two-dimension and three-dimension space). Also discussed in Chapter 2 is the problem that is being solved on the GPU and background mathematics needed for target generation. Chapter 3 discusses GPUs in general, CUDA C and the best practices for programming in CUDA C. The scenario using only a CPU and the scenario using both GPU and CPU are given in Chapter 4. Also, Chapter 4 briefly discusses small and large scale target trajectories, large-scale motion models (LSMM) and path constrained (with rotational matrix) and unconstrained targets. Chapter 4 also describes how the algorithm is implemented on the CPU and how it is implemented on the GPU, and answers why in the GPU is one motion leg, instead of one target, being generated at a time, and why not have each target generated in parallel. Chapter 4 ends with the breakdown of the CUDA kernel calls and the importance of the prefix sum (scan) technique. Chapter 5 presents simulation results that demonstrate the effectiveness of the proposed approaches. Conclusions, future work and applications are presented in Chapter 6.

1.1 Related Publication

- Y. Dinath, T. Kirubarajan, R. Tharmarasa, P. Valin and E. Meger, "Simulation of Large-Scale Multitarget Tracking Scenarios Using GPUs", Proceedings of the Signal and Data Processing of Small Targets, Baltimore, MD, May 2012.

Chapter 2

Target Generation

2.1 Targets

Targets are not limited to one type of object or thing. "Target" is applied ubiquitously and to any object that demonstrates a dynamic motion; hence, the target is represented by a point particle (infinitesimal dimensions) occupying a point in space. The target's dynamic motion helps identify what type of object the target is (Tewari, 2007). Some examples of targets are:

- Dismounted Combatants/Pedestrians: They have more mobility and have fewer constraints, even when being restricted to paths. Pedestrians move with relatively slow speeds, but can change direction quickly and without significant change in acceleration or speed. Pedestrians are able to be modeled in either a 2D or 3D environment.

- Automobiles: Vehicles are more restricted to lanes and have speed limitations, although they travel much faster than pedestrians. An automobile's motion can be limited to acceleration/deceleration, constant velocity and constant turn. Automobiles are able to be modeled in either a 2D or 3D environment.

- Watercraft: There are wide varieties of watercraft, such as ships, boats and submarines. Watercraft parameters, such as maximum speed, acceleration and mobility, change depending on the type of watercraft. Watercraft can be modeled in a 2D environment using a local coordinate system. For long distances, it should be modeled in 3D to account for the earth's curvature.

- Fixed-wing Aircraft: There are wide varieties of fixed-wing aircraft that usually vary in size. To account for the earth's curvature and because of the distances and flight dynamics usually involved with aircraft, it is best that aircraft motion be modeled only in a 3D environment using a global coordinate system.

If the target's physical properties were considered, then the shape and size of the target, its rotational motion and momentum changes would become important when modelling a target. However, the focus of this thesis is on the simulation of target trajectory, and the physical properties of said target are irrelevant. Therefore, this leaves a target with either two or three degrees of freedom depending on the environment. It can be assumed that the point particle is the target's centre of mass. (Galben, 2011)

A target's motion model (refer to Section 2.3) describes how the target moves through space with respect to time (Li and Jilkov, 2003).

Since the target is a point mass, it can be assumed that (Galben, 2011):

- the target is rigid

- the target rotates with the same angular speed

- there is no slippage - no loss of traction between the surface and target

2.2 Coordinate Systems

For the types of targets that will be looked at in this thesis, mainly (refer to Section 2.1) pedestrians, automobiles, watercraft and fixed-wing aircraft, their movements would be best described in either the Cartesian coordinate system (refer to Subsection 2.2.1) or geographic coordinate system (refer to Subsection 2.2.2). The ideal coordinate system depends on whether the target moves in either 2D or 3D space and the distance the target has to traverse. If the target does not travel a significant distance, then a local coordinates in 2D would be acceptable; otherwise, 3D space is the preferred environment.

2.2.1 Cartesian Coordinates System

The Cartesian coordinate system is a typical coordinate system for both 2D and 3D space, where X, Y and Z, as per convention, are orientated relative to each other using the right-hand rule. When generating, measuring or tracking targets, the Cartesian coordinate system is generally used to define the position of pedestrians and automobiles. The position of watercraft can be defined in the Cartesian coordinate system on a small scale, where the curvature of the earth would not be a factor.

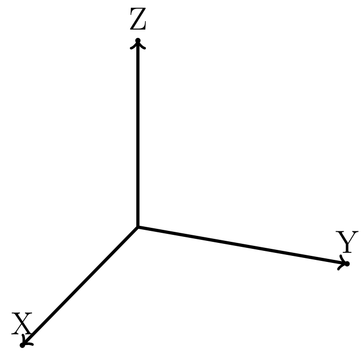

Figure 2.1: Cartesian coordinate system defined by the right-hand rule

2.2.2 WGS-84

The World Geodetic System - 1984 (WGS-84) is the standard geodetic reference system for international civil aviation navigation, and has been adopted by the Council of the International Civil Aviation Organization (ICAO) (EUROCONTROL and IfEN, 1998). The purpose of this standard is to represent an oblate spheroid, or a rotationally symmetric ellipsoid (refered to as an ellipsoid for now on), as the earth, such that the error between the ellipsoid and the earth is minimized (ICAO, 2002). There have been other suggested standards in the past such as WGS-60, WGS-66 and WGS-72. There have been significant changes to the standards due to the increase of satellite information, and the most recent changes to WGS-84 occured in 2004 (ICAO, 2002).

WGS-84 is an earth-fixed global reference frame. The origin is at the centre of the earth's mass. The x-axis is at the intersection of the meridian (0° longitude) and the equator (0° latitude), which the meridian and equator are defined by Bureau International de l'Heure (BIH). The y-axis is perpendicular counter clockwise (looking down the z-axis) along the equator from the x-axis. The z-axis is orthogonal to the x-axis and y-axis pointing up. However, due to polar motion, the z-axis is not a true representation of the earth's rotational axis. The x-axis, y-axis and z-axis satisfy the right-hand rule.

The WGS-84 ellipsoid, as shown in Figure 2.2, is defined as (from (ICAO, 2002)):

- Semi-major axis: $a = 6\ 378\ 137m$
- Ellipsoidal flattening (or oblateness): $f = \frac{1}{298.257223563}$
- Semi-minor axis: $b = a(1 - f) = 6\ 356\ 752m$
- Frist eccentricity: $e = \sqrt{1 - \frac{b^2}{a^2}} = 0.08181919$

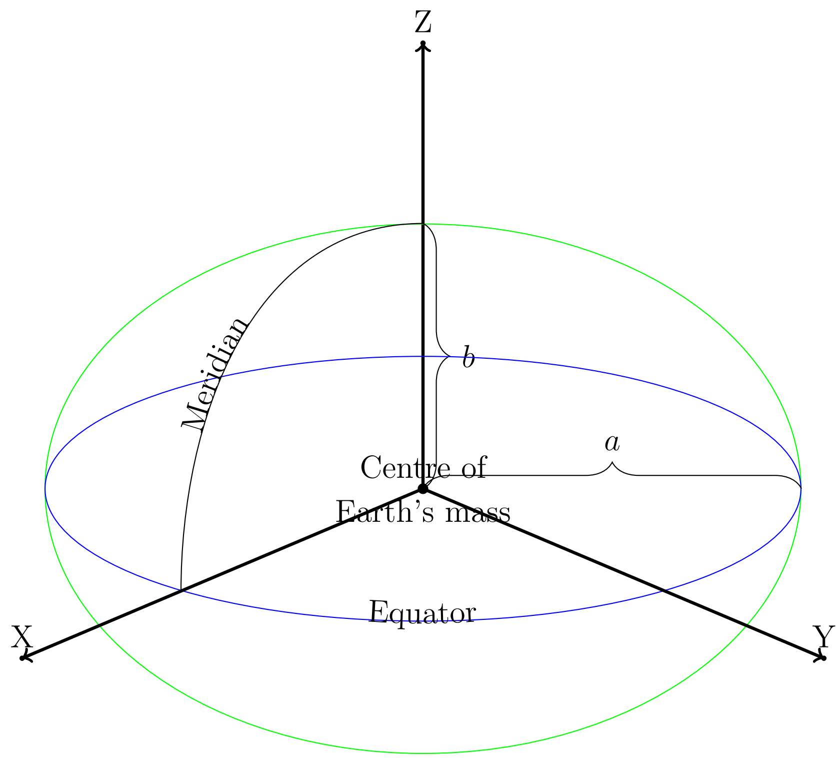

Figure 2.2: WGS-84 Coordinate system definition

2.2.3 ECEF

Earth-Centered, Earth-Fixed is a Cartesian coordinates system (refer to Subsection 2.2.1), where the origin, (0,0,0), represents the earth's centre of mass. Similar to WGS-84 (refer to Subsection 2.2.2), the x-axis is through the meridian and the equator, the y-axis is perpendicular counter clockwise (looking down the z-axis) horizontally from the x-axis, and the z-axis is orthogonal to the x-axis and y-axis pointing up. The x-axis, y-axis and z-axis satisfy the right-hand rule convention. (EUROCONTROL and IfEN, 1998)

2.3 Motion Models

The state equation is assumed to be of the following form (Li and Jilkov, 2003):

$$\mathbf{x_k} = F_k \mathbf{x_{k-1}} + G_k \nu_k \tag{2.1}$$

where, at time step k, $\mathbf{x_k}$ is the current state, F_k is the state transition matrix, G_k is the gain matrix for the independent white noise vector, ν_k.

2.3.1 Non-Maneuvering Targets

The Constant Velocity model is a well-known model for non-maneuvering targets (Li and Jilkov, 2003):

$$\mathbf{x}_k = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \end{bmatrix} \qquad F_k = \begin{bmatrix} 1 & T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad G_k = \begin{bmatrix} \frac{T^2}{2} & 0 \\ T & 0 \\ 0 & \frac{T^2}{2} \\ 0 & T \end{bmatrix} \tag{2.2}$$

where x and y are the standard Cartesian coordinates in 2D space, $\dot{x}$ and $\dot{y}$ represents the velocity along each dimension and T is the target motion sample duration. T is defined as the time difference between k and $(k-1)$.

2.3.2 Maneuvering Targets

The Constant Turn model is also a well-known model but for maneuvering targets (Li and Jilkov, 2003):

$$\mathbf{x_k} = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ \omega \end{bmatrix} \quad F_k = \begin{bmatrix} 1 & \frac{\sin\omega T}{\omega} & 0 & \frac{-(1-\cos\omega T)}{\omega} & 0 \\ 0 & \cos\omega T & 0 & -\sin\omega T & 0 \\ 0 & \frac{1-\cos\omega T}{\omega} & 1 & \frac{\sin\omega T}{\omega} & 0 \\ 0 & \sin\omega T & 0 & \cos\omega T & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad G_k = \begin{bmatrix} \frac{T^2}{2} & 0 & 0 \\ T & 0 & 0 \\ 0 & \frac{T^2}{2} & 0 \\ 0 & T & 0 \\ 0 & 0 & T \end{bmatrix} \tag{2.3}$$

where ω is the turn rate.

Other models that can be used are: Weiner Sequence Acceleration, Singer Acceleration, White Noise Jerk, Constant Turn with Cartesian Velocity, found in (Li and Jilkov, 2003). It is important to note that turn models are dependent on ω, and ω changes after each time step because of noise.

2.3.3 3D Models

For watercraft and aircraft type targets that traverse long distances, it is best to model the targets using the WGS-84 stantard earth model (refer to Subsection 2.2.2).

The motion can be broken down into two components: motion in the vertical plane and motion in the horizontal plane. Using the equations from (Galben, 2011):

$$
\begin{aligned}
x_k = x_{k-1} &- \underbrace{\left[R^m_{k-1}\cos\phi_{k-1} - R^m_k\cos\left(\phi_{k-1}+\dot{\phi}_k T\right)\right]\cos\lambda_{k-1}}_{\text{Vertical Plane}} \\
&- \underbrace{\left[R^v_{k-1}\cos\lambda_{k-1} - R^v_k\cos\left(\lambda_{k-1}+\dot{\lambda}_k T\right)\right]}_{\text{Horizontal Plane}} \\
y_k = y_{k-1} &- \underbrace{\left[R^m_{k-1}\cos\phi_{k-1} - R^m_k\cos\left(\phi_{k-1}+\dot{\phi}_k T\right)\right]\sin\lambda_{k-1}}_{\text{Vertical Plane}} \\
&- \underbrace{\left[R^v_{k-1}\sin\lambda_{k-1} - R^v_k\sin\left(\lambda_{k-1}+\dot{\lambda}_k T\right)\right]}_{\text{Horizontal Plane}} \\
z_k = z_{k-1} &- \left[R^m_{k-1}\sin\phi_{k-1} - R^m_k\sin\left(\phi_{k-1}+\dot{\phi}_k T\right)\right]
\end{aligned}
$$

where R^v is the radii of curvature in the prime vertical, R^m is the radii of curvature in the meridian, and $\dot{\phi}$ and $\dot{\lambda}$ are the latitude and longitudinal velocity, respectively, in radians per unit time. From (Sankowski, 2011)), R^v, R^m, $\dot{\phi}$ and $\dot{\lambda}$ are defined as:

$$R_k^v = \frac{a}{\sqrt{1 - e^2 \sin^2 \phi_k}}$$
$$R_k^m = \frac{a(1 - e^2)}{(1 - e^2 \sin^2 \phi_k)^{\frac{3}{2}}}$$

$$\dot{\lambda}_k = \frac{\upsilon_k^\lambda}{R_k^v \cos \phi}$$
$$\dot{\phi}_k = \frac{\upsilon_k^\phi}{R_k^m}$$

υ_k^ϕ and υ_k^λ is the ground velocity for latitude and longitude, respectively.

Figure 2.3 displays an example of the Earth, where a target is starting at the coordinate at $(x_{k-1}, y_{k-1}, z_{k-1})$ and following a great circle to the coordinate at (x_k, y_k, z_k).

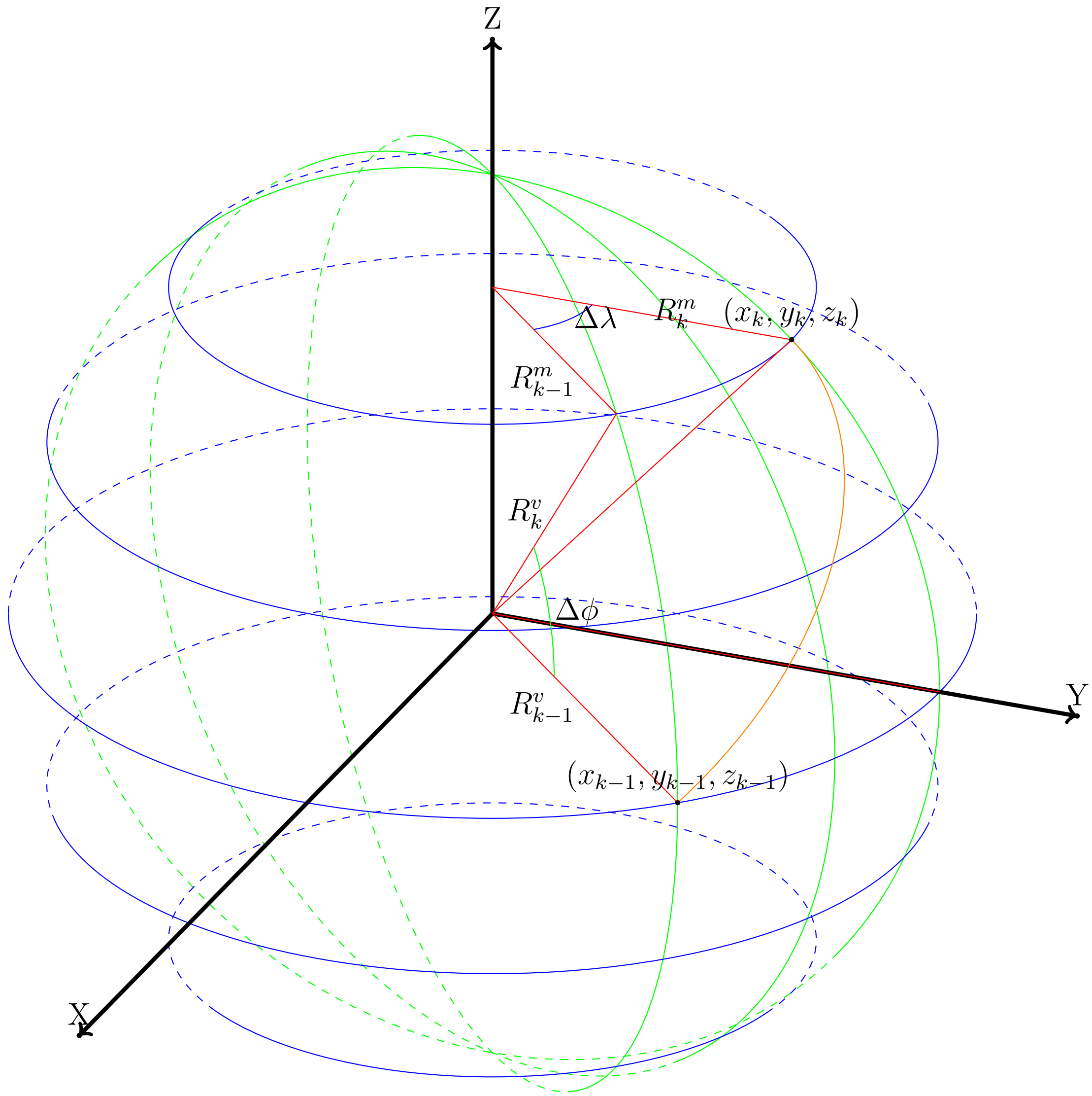

Figure 2.3: Spherical motion of a target from $k-1$ to k (Galben, 2011).

2.4 Coordinate Transformation

Using the WGS-84 ellipsoid constants from Subsection 2.2.2:

$a = 6\ 378\ 137$

$b = 6\ 356\ 752$

$e = 0.08181919$, and

h is the altitude, ϕ is the latitude and λ is the longitude.

2.4.1 Converting from WGS-84 to ECEF

$$x = (R^v + h)\cos\phi\cos\lambda$$
$$y = (R^v + h)\cos\phi\sin\lambda$$
$$z = (R^v \cdot (1 - e^2) + h)\sin\phi$$

where h, ϕ and λ are givens.

2.4.2 Converting from ECEF to WGS-84

$$\phi = \arctan\frac{z + \frac{a^2-b^2}{b}\sin^3\left(\arctan\frac{az}{b*\sqrt{x^2+y^2}}\right)}{\sqrt{x^2+y^2} - ae^2\cdot\cos^3\left(\arctan\frac{a*}{b\sqrt{x^2+y^2}}\right)}$$

$$\lambda = \arctan\frac{y}{x}$$

$$h = \begin{cases} \frac{\sqrt{x^2+y^2}}{\cos\phi} - R^v, & \text{if } |x| \geq 1 \quad | \quad |y| \geq 1 \\ |z| - b, & \text{o.w.} \end{cases}$$

h is corrected for numerical instability in altitude near exact poles if $|x| < 1$ & $|y| < 1$

2.5 Process Noise

Random noise is generated according to the normal distribution. The noise should be zero-mean, white and mutually independent (Bar-Shalom and Li, 1995). The total noise added to the state is

$$\Gamma_k = G_k \nu_k \tag{2.4}$$

where ν_k, the white noise vector for each time step, and has zero-mean and Q covariance (refer to Sections 4.2 and 4.3). Γ will take the form of

$$\begin{bmatrix} \Gamma_x & \Gamma_y & \Gamma_{x'} & \Gamma_{y'} \end{bmatrix} \qquad \begin{bmatrix} \Gamma_x & \Gamma_y & \Gamma_{x'} & \Gamma_{y'} & \Gamma_\omega \end{bmatrix}$$

for non-maneuvering and maneuvering targets, respectively.

2.6 Target Generation Theory

The state vector $\mathbf{x_k}$ (refer to Section 2.3), is derived using Equation (2.1). Note that the goal is to derive $\mathbf{x_k}$ such that it is optimal for parallel implementation on the GPU. This will be important in Subsection 4.5.2, when the code for the kernels is being developed.

First, find the state for every time step. The state for each time step in general is:

$$
\begin{aligned}
\mathbf{x_1} &= F_1\mathbf{x_0} + \Gamma_1 \\
\mathbf{x_2} &= F_2\mathbf{x_1} + \Gamma_2 \\
&\vdots \\
\mathbf{x_k} &= F_k\mathbf{x_{k-1}} + \Gamma_k
\end{aligned} \tag{2.5}
$$

where $G_k * \nu_k = \Gamma_k$.

Substituting and expanding x_1 into x_2, until x_{k-1} has been substituted into x_k

$$
\begin{aligned}
\mathbf{x_1} &= F_1\mathbf{x_0} + \Gamma_1 \\
\mathbf{x_2} &= F_2F_1\mathbf{x_0} + F_2\Gamma_1 + \Gamma_2 \\
&\vdots \\
\mathbf{x_k} &= \underbrace{F_k \cdots F_1\mathbf{x_0}}_{\text{State component}} + \underbrace{F_k \cdots F_2\Gamma_1 + \ldots + F_k\Gamma_{k-1} + \Gamma_k}_{\text{Noise component}}
\end{aligned} \tag{2.6}
$$

Equation (2.6) is broken down into two parts to show how the current state $\mathbf{x_k}$ is formed. State transition matrices are multiplied with the initial state to get the state component of $\mathbf{x_k}$, and with the noise vectors to get the noise component of $\mathbf{x_k}$.

Equation (2.6) can be generalized to:

$$
\mathbf{x_k} = \underbrace{\left[\prod_{i=0}^{k-1} F_{k-i}\right]\mathbf{x_0}}_{\text{State component}} + \underbrace{\left[\sum_{i=2}^{k}\left[\prod_{n=0}^{k-i} F_{k-n}\right]\Gamma_{i-1}\right] + \Gamma_k}_{\text{Noise component}} \tag{2.7}
$$

By multiplying and dividing the noise component, of Equation (2.6) or Equation

(2.7), with $F_k \cdots F_1$, helps to parallelize the state equation.

$$(F_1)^{-1}\Gamma_1 + \ldots + (F_{k-1} \cdots F_1)^{-1}\Gamma_{k-1} + (F_k \cdots F_1)^{-1}\Gamma_k \tag{2.8}$$

The general state equation, Equation (2.7), is updated to

$$\mathbf{x_k} = \left[\prod_{i=0}^{k-1} F_{k-i}\right] \left[x_0 + \sum_{n=1}^{k} \left[\left[\prod_{m=0}^{n-1} F_{n-m}\right]^{-1} \Gamma_n\right]\right] \tag{2.9}$$

The state vector of a target, $\mathbf{x_k}$, is broken up into two vectors

$$\mathbf{x_k} = \begin{bmatrix} \mathbf{x_k^1} & \mathbf{x_k^2} \end{bmatrix}' \tag{2.10}$$

where $\mathbf{x_k^1}$ is the state in regular Cartesian coordinates (refer to Subsection 2.3.1), and $\mathbf{x_k^2}$ is the non-Cartesian elements, such as ω (refer to Subsection 2.3.2).

It is assumed that F_k is not dependent on $\mathbf{x_k^1}$, and F_k is a function of the target motion sampling duration T. If F_k is dependent on any elements in $\mathbf{x_k^2}$, then F_k becomes nonlinear.

Notice that because of the structure of the F_k matrices that are used in this thesis, ω does not change when the previous state is being multiplied with the F_k matrix.

Since noise is being added to $\mathbf{x_k^2}$ at each time step it becomes cumulative. Therefore, the noise added to $\mathbf{x_k^2}$ at time k is

$$\sum_{i=1}^{k} \Gamma_i \tag{2.11}$$

or the sum of each Γ from every previous time step.

Chapter 3

GPU Programming

3.1 GPU

The Graphical Processing Unit (GPU) was initially designed to calculate values for computer graphics, so as to take an immense load off the CPU. This left the CPU available to perform its other necessary calculations and to control logic flow. Now, every modern personal computer (PC) has a GPU. A GPU that exceeds the parallel processing power of a CPU; [processing power that is now being sought by the scientific community, and being applied to applications other than just computer graphics. (Fung and Mann, 2004)

A General Purpose Graphical Processing Unit (GPGPU) is when the GPU is used for a general purpose, other than its typical processes, such as for computer graphics (J.D. Owens, 2005). However, there are trade-offs and bottlenecks that need to be considered. Is it worth the cost (in terms of time) to send data to and from the CPU to the GPU? Can a sequential task be sufficiently parallelized such that it can run effectively and efficiently on a GPU? (Fung and Mann, 2004). It has been demonstrated in the scientific community that a GPU performing computationally expensive tasks does so more efficiently than a CPU. The GPU is designed for intensive computation, so it is imperative that the CPU and GPU work together when parallelizing a program (Kirk and Hwu, 2010).

Although GPUs are now common, it is still difficult to program them effectively. Sequential algorithms must be redesigned to be parallel and algorithms have to be scalable.

3.2 CUDA

Compute Unified Device Architecture (CUDA), developed by NVIDIA, is an architecture for GPGPU programming. NVIDIA created their own set of rules and extensions that allow developers to write programs using the common C Programming language. (NVIDIA, 2011)

CUDA has a scalable parallel programming model that can be developed in a C/C++ programming environment. The programming environment is heterogeneous, meaning it allows the user to code both CPU and GPU (serial and parallel) in a single environment. The CUDA C programming environment allows the programmer to let the parallel algorithms be their primary concern, and not have to worry about the backend. Also, CUDA C programs are scalable, allowing the algorithm to be used on many different GPUs (NVIDIA, 2011).

3.3 CUDA C Programming

Some definitions are important when programming in CUDA C. The host is the CPU and the device is the GPU. Code is broken up into two parts: host code and device code. The host calls the device by calling a kernel function. Only code in the kernel function is executed in parallel. Mandatory inputs for the kernel are the number of blocks and the number of threads, and shared memory is an optional input (NVIDIA, 2011).

Inside a grid is a group of blocks, and inside each block is a group of threads (NVIDIA, 2011). A group of 32 threads is called a warp. It is important to have multiples of 32 threads inside each block for performance reasons. It is also important that each thread inside a warp do exactly the same thing (no branch divergence)(NVIDIA, 2012).

Blocks are independent, and run asynchronously, but they can never be synchronized within the kernel. Blocks have to be synchronised outside the kernel. The block's independence is what makes programs scalable. Threads run asynchronously, but can be synchronized within the kernel at the cost of time. (NVIDIA, 2011)

3.3.1 Memory

The CPU and GPU have different memory spaces, and data is transferred through the PCIe bus using CUDA APIs (NVIDIA, 2011).

There are a couple of important types of memory (not a complete list) (from (NVIDIA, 2011))

Global memory:

- resides in device memory

- shared between blocks
- the size of a block can be 32, 64 or 128 bytes
- blocks are aligned into multiples of 32, 64 or 128 bytes

Shared memory:

- much faster accessing than global memory, and best to replace global with shared memory in the kernel
- each block has its own shared memory
  - shared memory in one block cannot be accessed in a different block
- best for each thread to access different banks, or
- best for every thread to access one bank

Using page-locked or pinned host memory eliminates any need for CPU and GPU data transferred. Page-locked memory is a limited resource and only should be used when there is a need to transfer large amounts of data to and from the host and device (NVIDIA, 2011).

3.3.2 Toolkit

Initially released in June 2007, the toolkit has gone through thirteen other releases to date, which have included many advances to make parallel programming easier and faster. Most recently, NVIDIA have released a new complier that makes CUDA programs faster. (NVIDIA, 2011)

The toolkit is what enables developers to program in C/C++ to create GPU-accelerated programs. It includes a debugger and tips to optimize, guides and API reference (NVIDIA, 2011).

3.3.3 Libraries

There are many different GPU-accelerated libraries available for CUDA, such as fast Fourier transform library, sparse matrix library and others for high performance math routines (NIVIDA, 2012).

The runtime APIs are implemented in the cudart dynamic library. Some of the typical functions found in the cudart library are: **cudaGetDeviceProperties()**, which gets the GPU properties; **cudaMalloc()** and **cudaFree()**, which allocate and deallocate memory; and **cudaMemcpy()**, which transfers memory to and from the host and device, and also transfers between different variables on the device. **cudaHostAlloc()** and **cudaFreeHost()** allocates and frees, respectively, page-locked memory. **cudaHostRegister()** page-locks a range of memory allocated by **malloc()** (NVIDIA, 2011).

The library that is used in Section 4.5 is the CURAND Library. This library is used to generate a large set of normally distributed numbers, with zero mean and a standard deviation of one (NVIDIA, 2011).

3.4 CUDA and the Best Practices

The CUDA C Best Practices Guide has highlighted some important issues to consider when programming on the GPU (NVIDIA, 2012). It is important to consider these issues when programming on CUDA.

One of the most important things to do before coding is to identify the different ways to parallelize sequential code. If there are not many parts of the code that can be parallelized, it may not be worth it to parallelize a small algorithm. One reason is because of the bandwidth. The bandwidth is defined as the rate of data transfer, and can be used as a performance measurement. It is very costly, in terms of time, to transfer data from the host to the device and then back again. Therefore, the number of host and device transfers must be minimized; otherwise, there may be no gains in performance when executing code on the device compared to the host. Once the memory is on the device, what type of memory is used and how it is accessed is important. For example, it is always preferable to use shared memory over global memory, because it is more expensive to access global memory than it is to access shared memory. It is advised to copy memory from global memory into shared memory to avoid calling global memory more times than necessary (NVIDIA, 2011).

Because of how the GPU is designed, it is best to have the number of threads in a block of 32 for efficiency. As well, it is important to have all the threads in the warp execute the same path; otherwise, thread divergence happens and will increase the simulation run time (NVIDIA, 2011).

A few other miscellaneous items to keep in mind are: use unsigned integers as loop counters, shift operations for multiplying and dividing and use “#pragma unroll” for loops (NVIDIA, 2011).

Chapter 4

CPU and GPU Simulation

4.1 Target Trajectories

In order to create a simulation there are a few main parameters that need to be defined, such as

- The number of Monte Carlo runs (M_m)
- The number of targets (L) (can be randomly generated at the start)
- The number of motion legs of a target (M_L) (can also be randomly generated at the start)
- The initial state, x_0, for each target
- The F, G, (refer to Section 2.3) and the process noise covariance (refer to Section 2.5) for each motion leg

Each Monte Carlo run is independent of each other. Each target is independent of each other. However, elements in a motion leg are not independent of each other because of noise. With this, it is possible to generate the state for each target at every time step. For both the CPU and GPU, the processing power varies with M_m, L, M_L and the number of elements in each motion leg.

When implementing code for target generation on the GPU, two approaches can be taken. The first approach, the one implemented in this thesis (refer to Section 4.5), is to generate the entire motion leg for every target in parallel. The second approach is to generate every target in parallel. When writing a parallel algorithm, there are various things to consider, such as the number of targets and the number of elements in each motion leg.

For it to be possible to generate the entire motion leg in parallel on the GPU two assumptions must be made. The first assumption is that F_k can only be a function of $\mathbf{x_k^2}$ and not of $\mathbf{x_k^1}$ (refer to Equation (2.10)). The second assumption is that any element in $\mathbf{x_k^2}$, such as a turn rate, will not change when the F_k matrix is multiplied with the state matrix and will only change because of noise added.

There are two possibles methods that the user can use to create large-scale motion models. The first method is with unconstrained targets, and the second method is by constraining targets.

4.2 Large-Scale Motion Models with Path Unconstrained Targets

Unconstrained targets have almost every parameter randomly generated, including, but not limited to the number of targets, the number of motion legs per target, the initial state of the target, and the type of motion that the target will have. Unconstrained targets do not have to follow any path and are free to move in any direction.

Process noise covariance for unconstrained non-maneuvering (NM) and maneuvering (M) targets, respectively

$$Q_{NM} = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}$$

$$Q_M = \begin{bmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & 0 \\ 0 & 0 & \sigma_\omega^2 \end{bmatrix}$$

4.3 Large-Scale Motion Models with Constrained Targets

Trajectories for constrained targets are generated at specific nodes, have to follow a path and stay within the boundaries defined by the path. They do not enjoy the same freedom as unconstrained targets. The process noise covariance for constrained targets is different from unconstrained targets. The process noise is rotated such that the variance is greater in the direction of the target's instantaneous heading.

Process noise covariance for path constrained non-maneuvering and maneuvering targets, respectively (T. Kirubarajan, 1998)

$$Q_{NM} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \sigma_o^2 & 0 \\ 0 & \sigma_a^2 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$Q_M = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \sigma_o^2 & 0 & 0 \\ 0 & \sigma_a^2 & 0 \\ 0 & 0 & \sigma_\omega^2 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where θ is the path heading, to which the motion model is matched. Process noise covariance for the road and orthogonal to the road is given by σ_a^2 and σ_o^2, respectively. $\sigma_a^2 >> \sigma_o^2$ because of higher uncertainty in the direction of motion (T. Kirubarajan, 1998).

4.4 Approach 1: The CPU Algorithm

The CPU approach is a simple sequential algorithm. Find the state at each time step for each target for every Monte Carlo run, using (2.1). Figure 4.1 shows the CPU approach to calculate the state for every time step: one element per step.

```
for (MCR=0; MCR<numberOfMonteCarloRuns; MCR++)
{
  for (NT=0; NT<numberOfTargets; NT++)
  {
    for (ML=0; ML<numberOfMotionLegs; ML++)
    {
      for (MLD=0; MLD<MotionLegDuration; MLD++)
      {
        state = F*previousState + G*noise // one element per time step
      }
    }
  }
}

```

Figure 4.1: Sequential pseudocode for the CPU algorithm.

4.5 Approach 2: The GPU Algorithm

For the GPU approach, it is not just simply a matter of taking the CPU approach and generating that in parallel. Refer to Section 3.4 for reasons why. Figure 4.2 shows the GPU approach to calculate the state for every time step: all at once.

```
for (MCR=0; MCR<numberOfMonteCarloRuns; MCR++)
{
  for (NT=0; NT<numberOfTargets; NT++)
  {
    for (ML=0; ML<numberOfMotionLegs; ML++)
    {
      state = F*previousState + G*noise // every element in the motion leg at once
    }
  }
}

```

Figure 4.2: Motion Leg Parallelization pseudocode for the GPU algorithm.

4.5.1 Parallel Prefix Sum (Scan) with CUDA

The scan with CUDA is an important technique to implement (2.9). Simply doing a scan with an array of k elements,

a_0	a_1	a_2	a_3	a_4	a_5	$\cdots$	a_k

returns

I	a_0	$a_0 \bigoplus ... \bigoplus a_1$	$a_0 \bigoplus ... \bigoplus a_2$	$a_0 \bigoplus ... \bigoplus a_3$	$a_0 \bigoplus ... \bigoplus a_4$	$\cdots$	$a_0 \bigoplus ... \bigoplus a_{k-2}$

where I is the identity matrix and $\bigoplus$ is the binary associative operator (Harris, 2008).

For example, if you have a vector of

$$[2 \quad 6 \quad 1 \quad 3 \quad 0 \quad 8]$$

and the operator is addition then the scan will return

$$[0 \quad 2 \quad 8 \quad 9 \quad 12 \quad 12]$$

If the operator is changed to multiplication then the scan will return

$$[1 \quad 2 \quad 12 \quad 12 \quad 36 \quad 0]$$

To implement the scan, refer to Figure 4.3. Every other operation is a simple addition or product applied to each element of the scan. Each row is another period in time, and operations on columns on the same row happen at the same time. The number of threads equal half the number of cells, so every two arrows that meet is one thread.

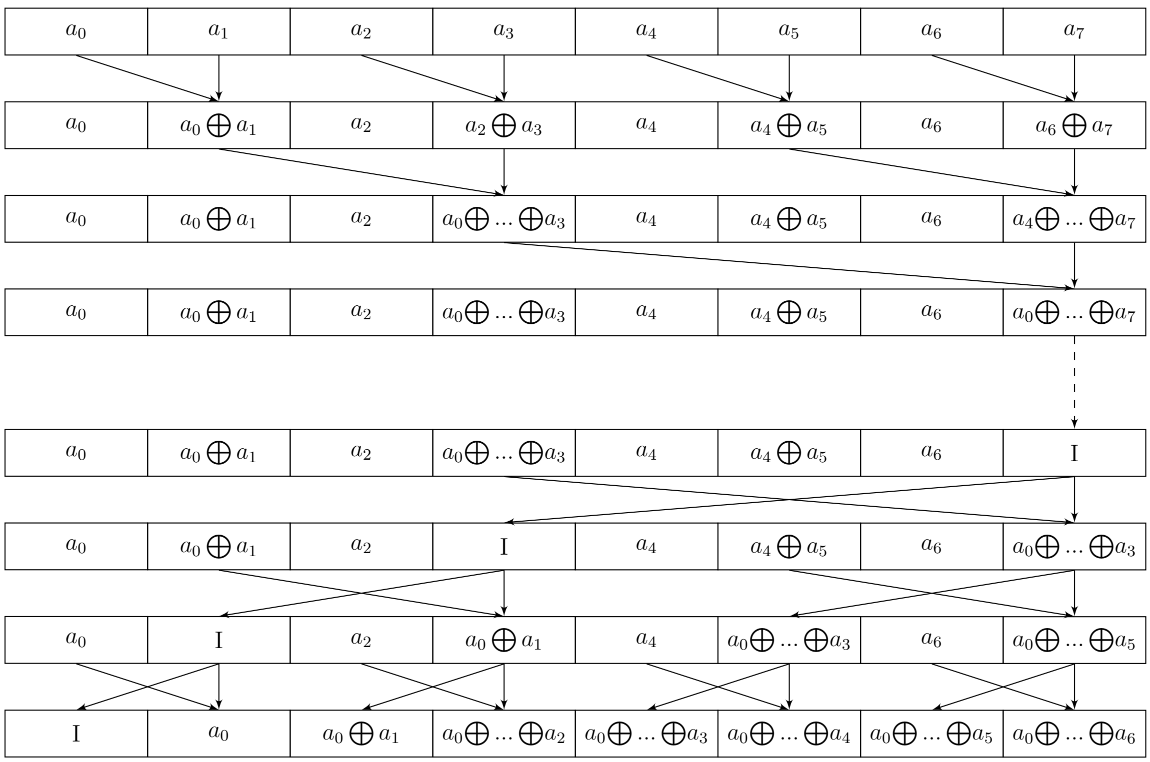

Figure 4.3: Work-efficient parallel scan, up and down sweep (Harris, 2008)

4.5.2 Kernel

In CUDA C, the kernel is a function that executes device code; invokes the device. Anything in the kernel gets compiled for the device and everything else is compiled for the host (Sanders and Kandrot, 2011).

The code needs to be broken down into four kernels. This is necessary when motion legs are broken up several times depending on how many GPU blocks are created. Blocks have to be synchronized, but this can only happen from within the host, and cannot happen from within the kernel.

To demonstrate why it is necessary to break down code into different kernels, take a vector with 16 elements (refer to Figure 4.4) for example. Each GPU block (2 blocks in this case) will process 8 elements. The first GPU block will process the top vector of elements, while the second GPU block will process the bottom vector of elements. With GPU blocks running asynchronously, it is not possible to know if the second GPU block will run before the first GPU block, or vise-versa. It is only possible to know when both GPU blocks are finished their task. Therefore, there must be two example kernels to complete the scan.

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}

Figure 4.4: Vector with 16 elements.

The first example kernel will perform the first half of the scan (refer to Figure 4.3). The results are shown in Figure 4.5

I	a_0	$a_0 \bigoplus ... \bigoplus a_1$	$a_0 \bigoplus ... \bigoplus a_2$	$a_0 \bigoplus ... \bigoplus a_3$	$a_0 \bigoplus ... \bigoplus a_4$	$a_0 \bigoplus ... \bigoplus a_5$	$a_0 \bigoplus ... \bigoplus a_6$
I	a_8	$a_8 \bigoplus ... \bigoplus a_9$	$a_8 \bigoplus ... \bigoplus a_{10}$	$a_8 \bigoplus ... \bigoplus a_{11}$	$a_8 \bigoplus ... \bigoplus a_{12}$	$a_8 \bigoplus ... \bigoplus a_{13}$	$a_8 \bigoplus ... \bigoplus a_{14}$

Figure 4.5: Scanned vector with 16 elements.

The by-product result of $a_0 \bigoplus ... \bigoplus a_7$ from the first GPU block, which would normally be discarded in the first half of the scan, is saved and used as an input for the second example kernel. $a_0 \bigoplus ... \bigoplus a_7$ is multiplied throughout the result of the second example kernel's second GPU block (results shown in Figure 4.6).

I	a_0	$a_0 \bigoplus ... \bigoplus a_1$	$a_0 \bigoplus ... \bigoplus a_2$	$a_0 \bigoplus ... \bigoplus a_3$	$a_0 \bigoplus ... \bigoplus a_4$	$a_0 \bigoplus ... \bigoplus a_5$	$a_0 \bigoplus ... \bigoplus a_6$
$a_0 \bigoplus a_7$	$a_0 \bigoplus a_8$	$a_0 \bigoplus ... \bigoplus a_9$	$a_0 \bigoplus ... \bigoplus a_{10}$	$a_0 \bigoplus ... \bigoplus a_{11}$	$a_0 \bigoplus ... \bigoplus a_{12}$	$a_0 \bigoplus ... \bigoplus a_{13}$	$a_0 \bigoplus ... \bigoplus a_{14}$

Figure 4.6: Scanned vector with 16 elements.

The scan is completed using two example kernels.

The rest of this section will show the breakdown of the four kernels used to implement the target generating method on the GPU.

Kernel 1

In the first kernel, for every element (or time step) the white noise matrix is multiplied by gain matrix, $G * \nu_k$, to get Γ_k (refer to Figure 4.7).

0	Γ_1	Γ_2	Γ_3	Γ_4	Γ_5	$\cdots$	Γ_k

Figure 4.7: Noise at every time step of a motion leg.

To get the cumulative noise added at each time step (refer to Figure 4.8), the prefix sum (scan) (refer to Subsection 4.5.1) will need to be performed on the result of Figure 4.7. However, the scan will be broken up into two kernels, because there is a possibility that the motion leg spans more than one GPU block. The second half of the scan will be completed in Kernel 2.

Kernel 2

In the second kernel, it is necessary to complete the scan started in Kernel 1. The second half of the scan is performed as to find the cumulative noise at each time step. Final results of the scan is displayed in Figure 4.8.

I	Γ_1	$\Gamma_1 + ... + \Gamma_2$	$\Gamma_1 + ... + \Gamma_3$	$\Gamma_1 + ... + \Gamma_4$	$\Gamma_1 + ... + \Gamma_5$	$\cdots$	$\Gamma_1 + ... + \Gamma_k$

Figure 4.8: The scan of Figure 4.7.

Note that the noise for each element (refer to 4.8) will only be added to the corresponding element of $\mathbf{x_k^2}$ as an input to the F-matrix (refer to Section 2.5). For example, Γ_ω is added to the ω element of $\mathbf{x_k^2}$. Since $\mathbf{x_k^2}$ is an input to F_k, it is now possible to find F_k (Figure 4.9). The noise that will be added to the state is calculated differently and at the end (refer to Kernel 3 and Kernel 4).

To solve the highlighted part of Equation (4.1) and to complete the second kernel, the first half of the scan is performed on the F-matrix vector (refer to Figure 4.9).

$$x_k = \left[\prod_{i=0}^{k-1} F_{k-i}\right] * \left[x_0 + \sum_{n=1}^{k}\left[\left[\prod_{m=0}^{n-1} F_{n-m}\right]^{-1} \Gamma_n\right]\right] \tag{4.1}$$

F_0	F_1	F_2	F_3	F_4	F_5	$\cdots$	F_k

Figure 4.9: F-matrix at every time step.

Kernel 3

The third kernel is the most complex. The third kernel starts by completing the F-matrix vector (refer to Figure 4.9) scan started in the second kernel, and the result is shown in Figure 4.10.

I	F_1	$F_2 ... F_1$	$F_3 ... F_1$	$F_4 ... F_1$	$F_5 ... F_1$	$\cdots$	$F_k ... F_1$

Figure 4.10: The scan of Figure 4.9.

Then, as per the highlighted part of Equation (4.2), the third kernel takes the inverse of every element in the F-matrix scanned vector, and the result is shown in Figure 4.11.

$$x_k = \left[\prod_{i=0}^{k-1} F_{k-i}\right] * \left[x_0 + \sum_{n=1}^{k}\left[\left[\prod_{m=0}^{n-1} F_{n-m}\right]^{-1} \Gamma_n\right]\right] \tag{4.2}$$

I	$(F_1)^{-1}$	$(F_2...F_1)^{-1}$	$(F_3...F_1)^{-1}$	$(F_4...F_1)^{-1}$	$(F_5...F_1)^{-1}$	$\cdots$	$(F_k...F_1)^{-1}$

Figure 4.11: Inverse of each element of the scanned F-matrix vector (Figure 4.10).

The noise vector, highlighted in Equation (4.3)

$$x_k = \left[\prod_{i=0}^{k-1} F_{k-i}\right] * \left[x_0 + \sum_{n=1}^{k}\left[\left[\prod_{m=0}^{n-1} F_{n-m}\right]^{-1} \Gamma_n\right]\right] \tag{4.3}$$

is from Figure 4.7, calculated in Kernel 1.

Multiply each element of the inverse F-matrix scanned vector (refer to Figure 4.11) with the noise at every time step vector (refer to Figure 4.7). This is also demonstrated in the highlight part of Equation 4.4.

$$x_k = \left[\prod_{i=0}^{k-1} F_{k-i}\right] * \left[x_0 + \sum_{n=1}^{k}\left[\left[\prod_{m=0}^{n-1} F_{n-m}\right]^{-1} \Gamma_n\right]\right] \tag{4.4}$$

To get the total noise added to the state for every time step, the result needs to go through a scan.

Kernel 4

The first step is to complete the scan from the end of Kernel 3. Add the total noise (refer to Equation 4.4) to the initial state and multiply it by the scanned F vector for every time step to get the complete motion of a target.

4.5.3 Kernel Inputs

The kernel processes one motion leg at time, so the amount of memory that is being used in the kernel changes dynamically with the number of elements in the motion leg.

The grid is one-dimensional (1D) and is determined by

$$\text{grid} = \frac{\text{motionLegDuration} + \text{block\_dim}}{\text{block\_dim}} \tag{4.5}$$

where *motionLegDuration* is the duration of a motion leg of a particular target, and *block_dim* is the first power of two greater than *motionLegDuration*.

The number of threads is equal to $\frac{\text{block\_dim}}{2}$, so that there are twice as many elements as there are threads.

The block dimension is determined by the first power of two greater than the motion leg duration. However, because shared memory per block is finite, the block dimension has to be reduced to a lower power of two. By reducing the block dimension, the motion leg will be split up and processed on multiple blocks.

it could be less than that, so that the shared memory per block is not exceeded. Shared memory per block is device dependent.

The shared memory for the kernel is an option input. However, it is used and it changes significantly depending on the size of the noise vector, F-matrix, G-matrix and state.

4.5.4 CUDA Best Practices Implemented in the Kernel

For implementation, each cell (in any figures above) is considered to be one time step, so each row of cells can be considered to be one motion leg. For the scan (refer to Subsection 4.5.1) to be implemented, there needs to be one thread for every two cells. This means that the number of threads is half of the CUDA block dimension. Since it is best to have multiples of 32 threads (32 threads is one warp), the block dimension will be multiples of 64.

Some other important steps implemented on the GPU for performance are:

- State equation is broken down into many parts so that each can be parallelized
- Every thread inside a warp performs identical calculations
- Input and output data is transferred from global memory to shared memory

Chapter 5

Results

In Section 2.6, a new general state equation was derived for parallel implementation of target generation. Subsection 4.5.2 showed how to parallelize the target generation by parallelizing the motion legs. This chapter will bring both Section 2.6 and Subsection 4.5.2 together and execute the approches discussed in Sections 4.4 and 4.5, and demonstrates the results of the implementation with two target generation scenarios on physical hardware. By providing an explanation and the significance of the results of the executed approches, this chapter will also discuss the benefits of using a GPU and CPU (in an integrated environment) compared to only using CPU for small and large scale target generation. The measurements that will be helpful when determining the benefits are execution time of CPU and GPU, the speedup factor, memory transfer time between the host and device and Giga Floating-Point Operations Per Second (GFLOPS). There will be two scenarios used to demonstrate the performance of the GPU and CPU.

The first scenario was a simple constant velocity model (refer to Subsection 2.3.1) with zero-mean normally distributed noise (refer to Section 2.5) that ran for 100

Monte Carlo runs. Every target's motion leg's duration increased, by increments of one second, from 1 second to 500 seconds so that it can be shown how the execution time varies with the motion leg duration. The output of the first scenario is the position (in Cartesian coordinates), velocity, acceleration and the time when a target was at a particular location.

The second scenario, a more practical one, is a 3D scenario of ships traversing around the earth. Multiple simulations were run where the number of targets varied so that it can be shown how the execution time varies with the number of targets. The simulation parameters were one Monte Carlo run, and either 10, 50, 100, 500 or 1000 targets, where each target moved at speeds between 2 to 15 metres per second, time steps were fixed to every 30 seconds and each target ran for 18000 seconds (5 hours). A similar scenario, but with 10000 targets, was used by exactEarth and a radar group from York University. The output of each simulation is the position (in geographic coordinates), ground velocity, ground acceleration and the time when a target was at a particular location.

Each scenario was run three times and then averaged; this was to eliminate any random fluctuation in the CPU and GPU results. The two scenarios are enough to demonstrate everything necessary to appreciate the results. Based on the setup of both CPU and GPU implementations, the first scenario can also be described as being 1 Monte Carlo run with 100 targets, each target having a motion leg that varies between 1 second to 500 seconds in duration. Changing the state transition matrix (refer to Section 2.3) from a constant velocity model, to something more complex, such as a constant turn model, would have no significant impact on the outcome of the results, because the process stays the same. The second scenario modifies the number of targets, rather than the duration of the motion leg. When comparing the two scenarios, note that for the first scenario the execution time is the amount of time it takes to complete all 100 Monte Carlo runs, whereas, the second scenario has only one Monte Carlo run. This thesis does not run 100 Monte Carlo runs for the second scenario because it takes too much time.

The GPU is a NVIDIA GTX 480, which has 480 cores (32 CUDA cores per multiprocessor by 15 multiprocessors).

The output data was first created using MATLAB as a quick was to see what the desired target states would be. Then the output data was created using both the CPU and GPU method in C. The data was compared with each other to ensure that the output data was correct.

5.1 Performance Measurements

5.1.1 Execution Time

Execution time is the time it takes for a program to perform its function from inputs to outputs. In CUDA, the execution time can be found using CUDA C commands. First the device must be synchronized before starting the timer and after ending the timer. Synchronization will ensure that the GPU has finished processing all threads and will be accurately timed. Figure 5.1 shows the code to accurately time the CPU's and GPU's execution time.

```
cudaThreadSynchronize();
cudaEventRecord(start, 0);
generateTrajectory(); //Either CPU or GPU
cudaThreadSynchronize();
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time,start,stop);
```

Figure 5.1: Sequential pseudocode for the CPU algorithm.

Execution time is arguably the simplest measurement to calculate, and the results can be easily interpreted. Execution time was used to test what improvement the GPU offers over using the CPU. Another example of using execution time was when testing various options for only the GPU, such as if using pinned memory would be beneficial over using allocated memory, or if there can be a reduction in the kernel run time if the number of threads are changed per kernel.

5.1.2 Speedup Factor

One way to determine the performance of the parallel algorithm is to find the speed up factor, which is defined as

$$S = \frac{t(\mathrm{CPU})}{t(\mathrm{GPU})} \tag{5.1}$$

where S is the speedup and $t(\cdot)$ is the execution time of either the CPU or the GPU.

The speedup factor is a manipulation of the execution time, so the benefits of determining the speedup factor are the same as the execution time. The speedup factor is not limited to Equation 5.1, it can also be used to find if one set up of the GPU is better than another setup for the GPU, i.e. $(\frac{t(\mathrm{GPU_o})}{t(\mathrm{GPU_n})})$. Where $\mathrm{GPU_o}$ is the old GPU method and $\mathrm{GPU_n}$ is the new GPU method.

$$S \quad \begin{cases} > 1 & \text{increase in performance (GPU or GPU}_{new}\text{ is better)} \\ < 1 & \text{decrease in performance (CPU or GPU}_{old}\text{ is better)} \\ = 1 & \text{no benefit of using the GPU or GPU}_{new} \end{cases}$$

5.1.3 GFLOPS

Giga Floating-Point Operations Per Second (GFLOPS) is the value of the number of the double-precision arithmetic rate of a GPU. The theoritical peak double-precision arithmetic rate for NVIDIA GTX 480 is approximately 168 GFLOPS.

GFLOPS is calculated using Equation (5.2)

$$\text{GFLOPS} = \frac{\text{Floating-points operations in kernel}}{\text{kernel execution time (seconds)}} 10^{-9} \tag{5.2}$$

It is practically impossible to get near theoretical value of GFLOPS for any GPU. To get any practical results out of the GPU, the kernel has to perform tasks other than floating-point operations, other tasks that get included in the execution time. Other tasks such as function calls, memory allocation and deallocation, input and output memory transfers and single-precision calculations in the kernel.

5.1.4 Bandwidth

The time to transfer data between the host and device is usually a bottleneck for smaller GPU programs. For NIVIDIA's GTX 480 the memory bandwidth is 177.4

GB/s. In both scenarios, only a relatively small amount of data is needed to generate a significantly amount of data. Therefore, the time it takes to transfer memory to the host from the device is much greater than the time is takes to transfer memory to the device from the host.

5.2 Scenario 1

Figure 5.2 shows the CPU and GPU execution time versus the duration of the target's motion leg.

The CPU's execution time had many fluctuation, however, there is a clear linear increase in the CPU's execution time. The CPU's growth is as expected, with every increment in the target's motion leg duration the execution time should increase at the same rate (slope is constant). Based on the CPU's setup, the CPU's execution time will still increase steadily even if any of the *for* loops were switched in Figure 4.1.

For majority of the GPU's execution time, it had a steady slope as time increased. The GPU growth can be modelled with two functions, first growing similar to a $f(x) = \sqrt{x}$ function for durations approximately less than 60 seconds in this simulation, and then increasing linearly approximately after durations of 60 seconds. There is a significant amount of overhead added to the GPU's exeuction time. However, as with economies of scale, the cost of overhead reduces as the motion leg duration increases.

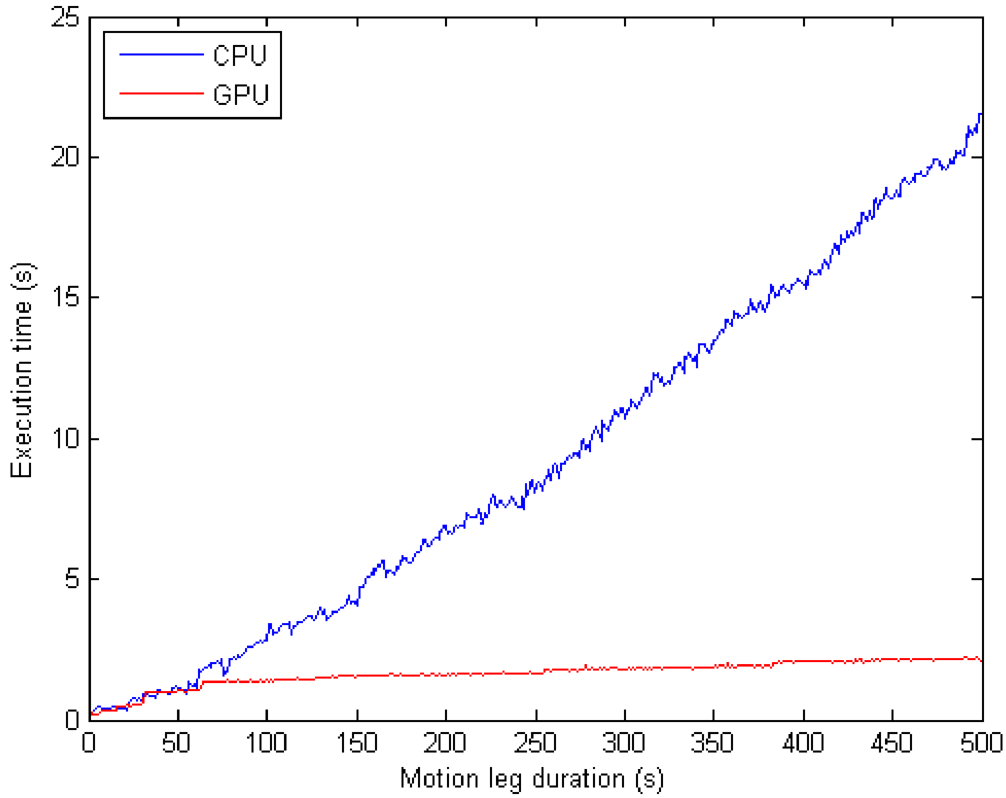

Figure 5.2: CPU and GPU execution time versus the duration of the motion leg for scenario 1.

Figure 5.3 demonstrates how as the target's motion leg duration increases the speedup factor also increases. However, there is a significant amount of fluctuation when the duration of the target's motion leg is small. Again, this is due to the overhead costs associated with initializing the GPU.

At low motion leg durations, it is not clear if the GPU is better than the CPU. However, as the duration of the motion leg increases beyond 60, it is clear the GPU is beneficial.

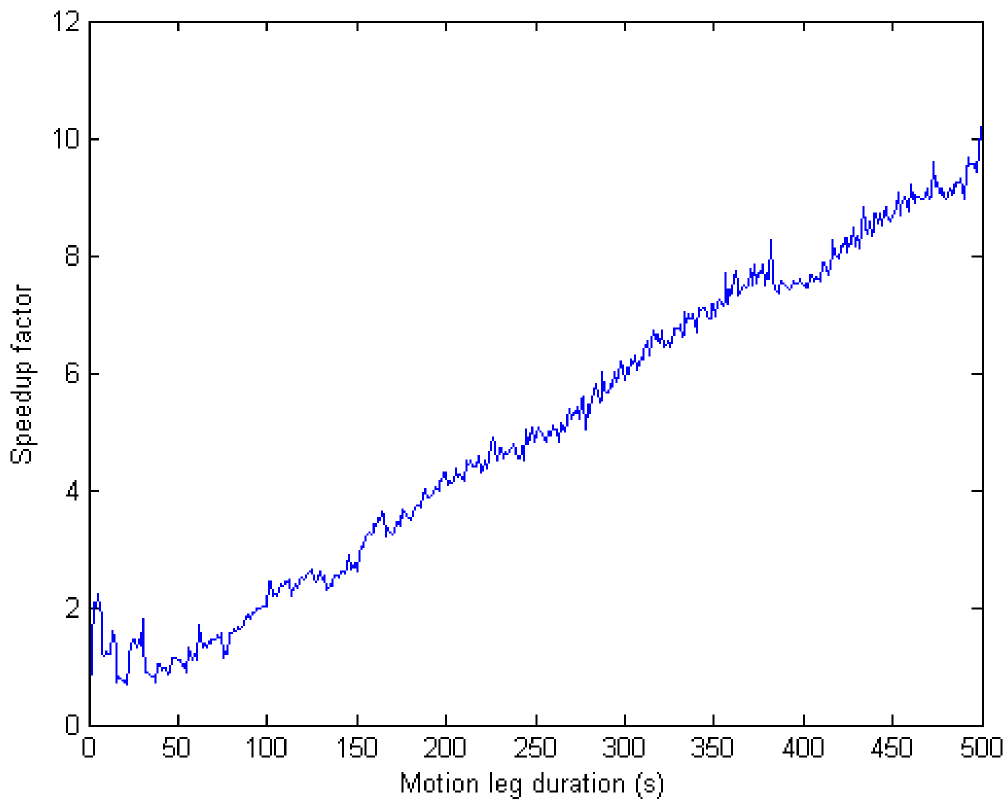

Figure 5.3: Speedup factor of the GPU over the CPU vs the Duration of the Motion Leg for scenario 1.

Tables 5.1 and 5.2 show the transfer time from host to device, device to host, the total of host to device and device to host, and the total execution time of the GPU for a particular duration. As shown, the execution time for the GPU is much greater than the total transfer time. Therefore, the transfer time between host and device is insignificant. It should be noted that time can be saved by using pinned memory, but it is only beneficial to use pinned memory for small GPU programs.

Duration (s)	H to D (μs)	D to H (μs)	Total Transfer (μs)	t(GPU) (μs)
10	1.824	6.144	7.968	40111
50	1.856	16.864	18.720	1008023
100	1.856	30.208	32.064	1380652
500	1.856	138.528	140.384	2196758
1000	1.824	280.576	282.400	2205157

Table 5.1: Transfer time using memcpy for scenario 1

Duration (s)	H to D (μs)	D to H (μs)	Total Transfer (μs)	t(GPU) (μs)
10	0.928	-	0.928	380321
50	0.928	-	0.928	1099486
100	0.928	-	0.928	1427561
500	0.928	-	0.928	2183549
1000	0.928	-	0.928	2259842

Table 5.2: Transfer time using pinned memory for scenario 1

Recall that the method to generate target trajectories in this thesis is by the motion leg. As the target's motion leg duration increases, so does the utilization of the GPU, because the motion leg duration is directly proportional to the number of blocks used. With more blocks doing more calculations in the same amount of time, there is an increase in GFLOPS. This is shown in Table 5.3, which displays the GFLOPS per kernel.

Duration (s)	Kernel 1	Kernel 2	Kernel 3	Kernel 4
10	0.141	1.002	2.157	0.285
50	0.215	2.494	3.614	0.406
100	0.338	4.280	6.167	0.816
500	1.135	15.642	17.538	2.402
1000	2.001	29.488	26.292	3.549

Table 5.3: GFLOPS per kernel

Refer to Subsection 4.5.2, the second kernel is the smallest, although performs mainly floating-point operations. The third kernel is the largest and has a large amount of memory transfers from global to shared memory, but also performs a lot of floating-point operations. The first and last kernel do not perform a extensive amount of floating-point operations, which means a lot of time is dedicated to other functions, such as memory transfers between global and shared memory, single-precision operations.

5.3 Scenario 2

Figure 5.4 displays the execution time of either GPU or CPU versus the number of targets. Note that data only exists for 1, 10, 50, 100, 500 and 1000 targets, everything else is interpolated. Both the GPU and CPU increase at a steady rate. The rate of increase for the GPU is much higher in this scenario than it is for the first scenario.

The rate of increase is higher because the GPU processes entire motion legs (and not all targets) at a time, and in this simulation the motion leg duration is fixed. Therefore, in this simulation, the GPU acts exactly like the CPU does, i.e. doubling the number of targets will double the execution time (all things being the same).

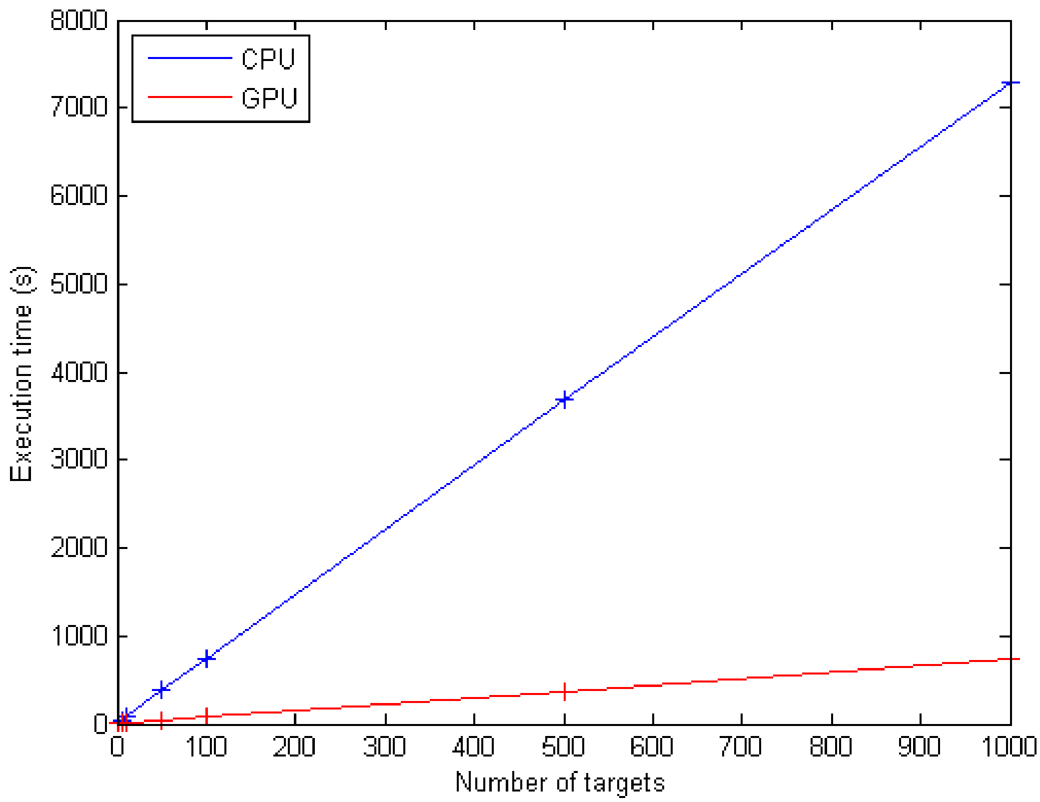

Figure 5.4: CPU and GPU execution time versus the number of targets for scenario 2.

Since the motion leg duration is fixed, so is the number of blocks per call (regardless of the number of targets). Therefore, the speedup factor is essentially fixed at 10 times, as shown in Figure 5.5.

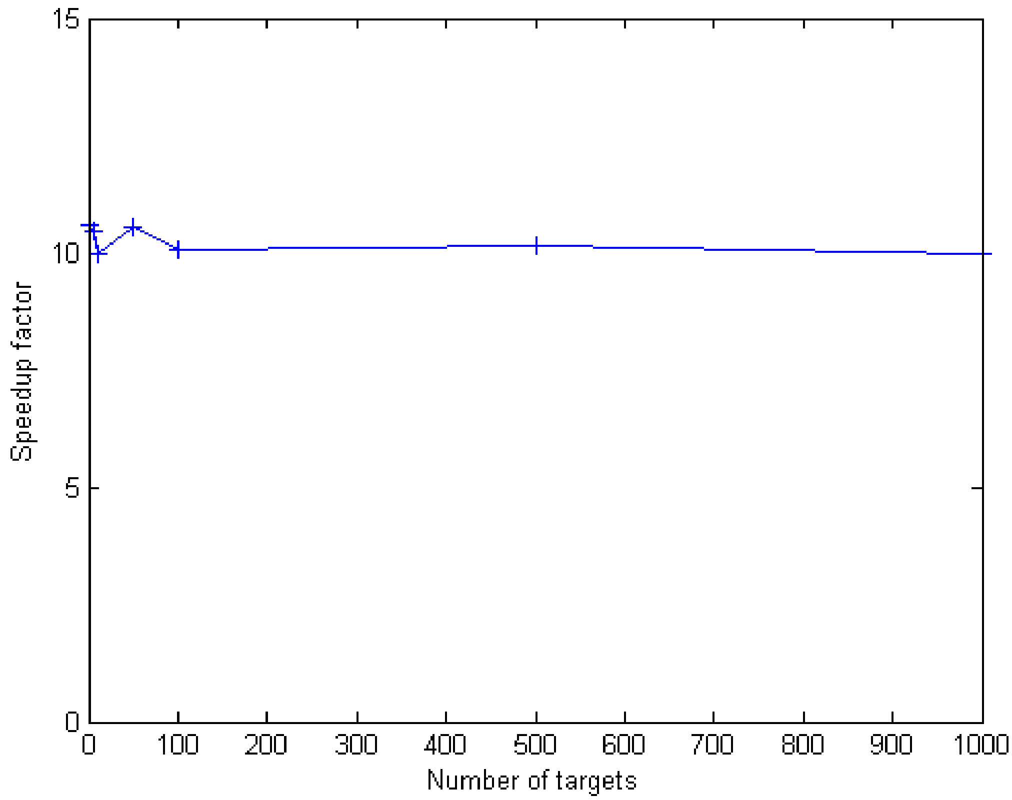

Figure 5.5: Speedup of the GPU over the CPU versus the number of targets for scenario 2.

In Table 5.4, as with Table 5.1, the time it takes to transfer memory to the device from the host (and vice-versa) is insignificant compared to the total time the GPU runs.

Targets	H to D (μs)	D to H (μs)	Total Transfer (μs)	t(GPU) (μs)
10	1.890	777	778.890	7331984
50	1.920	3970	3971.920	36482559
100	1.920	8860	8861.920	73054031
500	2.976	46100	46102.976	364412018
1000	4.256	84496	84500.256	729139404

Table 5.4: Transfer time using memcpy memory for scenario 2

The same reasoning as above is needed to explain the GFLOPS results in Table 5.5. Every target has to call all the kernels. As long as the motion leg duration is fixed, the GFLOPS per kernel will be the same regardless of the number of targets per simulation.

Targets	Kernel 1	Kernel 2	Kernel 3	Kernel 4
10	2.406	30.752	9.697	1.187
50	2.356	30.742	9.717	1.188
100	2.378	30.772	9.708	1.188
500	2.367	30.758	9.710	1.188
1000	2.377	30.756	9.708	1.188

Table 5.5: GFLOPS

5.4 Final Thoughts about Both Simulations

From profiling both applications, the majority of the time was spent in the third kernel, which was expected. Generating random numbers took an insignificant amount of time, especially when the duration of the motion leg was large. The number of divergent branches was less than 1% of the total number of branches.

Often the branches diverged when storing and accessing memory location. There are fewer locations than there are threads. Also, for the last block, when there is an excess number of elements, the program needs to prevent accessing elements greater than the duration.

Chapter 6

Conclusion and Future Work

This thesis demonstrates one approach to parallelize the generation of simulated target trajectories by using the GPU in conjunction with the CPU. The approach used was to parallelize the entire motion leg of a target at a time. To begin, it was necessary to define the problem. What is involved in generating target trajectories, both in 2D and in 3D, especially with the different coordinate systems. It was then necessary to understand the programming environment. Programming in parallel has become common because parallel hardware is cheaper. However, this does not mean that parallel programming has become easier. The state equation needed to be manipulated such that it could be used in developing a parallel algorithm; this required a few assumptions about the motion models.

The results of parallelizing the motion legs were significant from the beginning. However, there is room for improvement. The first way would be to reduce the amount of overhead prior to calling the GPU. Other ways include taking advantage of latest CUDA toolkits and newer generations of GPUs. Another option is to explore parallelization of targets instead of motion leg's time steps; having one target per block and breaking up the motion legs into threads. To run one target per thread is not advisable because of thread divergence.

Target generation is only the first step of many. User generated target trajectories is necessary in testing algorithms such as the ones found in target tracking or path detection. Having true data enables accurate comparisons between different trackers. With target trackers and path detection algorithms, there is an increasing need for large-scale target trajectories.

Bibliography

Bar-Shalom, Y. and Li, X. R. (1995). *Multitarget-Multisensor Tracking: Principles and Techniques*. YBS Publishing, Storrs, CT.

EUROCONTROL and IfEN (1998). *WGS 84 Implementation Manual*. European Organization for the Safety of Air Navigation, and Institute of Geodesy and Navigation, Brussels, Belgium; and University FAF Munich, Germany, 2.1 edition.

Fung, J. and Mann, S. (2004). Computer vision signal processing on graphics processing units. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages V93–V96.

Galben, G. (2011). New three-dimensional velocity motion model and composite odometry-inertial motion model for local autonomous navigation. *IEEE Transactions on Vehicular Technology*, **60**(3), 771–781.

Harris, M. (2008). *GPU Gems 3*, chapter Chapter 39. Parallel Prefix Sum (Scan) with CUDA. Addison-Wesley Professional, Upper Saddle River, NJ, 3 edition.

ICAO (2002). *World Geodetic System - 1984 (WGS 84) Manual*. International Civil Aviation Organization, Montreal, Canada, 2 edition.

J.D. Owens, D. Luebke, N. G. M. H. J. K. A. L. T. P. (2005). A survey of general-purpose computation on graphics hardware. *In Eurographics 2005, State of the Art Reports*, pages 21–51.

J.P. Holdren, E. L. and Varmus, H. (2010). Report to the president and congress - designing a digital future: Federally funded research and development in networking and information technology. Technical report, Executive Office of the President and President's Council of Advisors on Science and Technology.

Kirk, D. B. and Hwu, W. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers, Burlington, MA, 1 edition.

Li, X. R. and Jilkov, V. P. (2003). Survey of maneuvering target tracking. part 1: Dynamic models. *IEEE Transactions on Aerospace and Electronic Systems*, **39**(4), 1333–1364.

NIVIDA (2012). Cuda toolkit. http://developer.nvidia.com/cuda-toolkit.

NVIDIA (2011). *CUDA C Programming Guide*. NVIDIA, Santa Clara, CA, 4.2 edition.

NVIDIA (2012). *CUDA C Best Practices Guide*. NVIDIA, Santa Clara, CA, 4.1 edition.

Sanders, J. and Kandrot, E. (2011). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, Upper Saddle River, NJ, 1 edition.

Sankowski, M. (2011). Reference model of aircraft movements in geodetic coordinates. In *Radar Symposium (IRS), 2011 Proceedings International*, pages 874–879.

T. Kirubarajan, Y. Bar-Shalom, K. P. and Kadar, I. (2000). Ground target tracking with topography-based variable structure imm estimator. *IEEE Transactions on Aerospace and Electronic Systems*, **36**, 26–46.

T. Kirubarajan, Y. Bar-Shalom, K. P. I. K. B. A. E. E. (1998). Tracking ground targets with road constraints using an imm estimator. In *IEEE Aerospace Conference*, volume 5, pages 5–12.

Tewari, A. (2007). *Atmospheric and Space Flight Dynamics: Modeling and Simulation with MATLAB® and Simulink® (Modeling and Simulation in Science, Engineering and Technology)*. Birkhäuser Boston, Cambridge, MA, 1 edition.