GPU-Specific Kalman Filtering and Retrodiction for Large-Scale Target Tracking

GPU-SPECIFIC KALMAN FILTERING AND RETRODICTION FOR LARGE-SCALE TARGET TRACKING

BY

SEAN TAGER, B.Sc.

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 (2013) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	GPU-Specific Kalman Filtering and Retrodiction for Large-Scale Target Tracking
AUTHOR:	Sean Tager B.Sc., (Electrical Engineering) McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. T. Kirubarajan
NUMBER OF PAGES:	xii, 100

To my sister Lesley, who bought me my first novel

Abstract

In the field of Tracking and Data Fusion most, if not all, computations executed by a computer are carried out serially. The sole part of the process that is not entirely serial is the collection of data from multiple sensors, which can be executed in parallel. However, once the data is to be filtered the most likely candidate is a serial algorithm. This is due in large part to the algorithms themselves that have been developed over the last several decades for use on conventional computers that have been left void of parallel computing capabilities, until now. With the arrival of graphical processing units, or GPUs, the tracking community is in a favourable position to exploit the functionality of parallel processing in order to track a growing number of targets.

The problem, however, begins with the sheer labour of having to convert all the pre-existing serial tracking algorithms into parallel ones. This is clearly a daunting task when one considers the extent to which the tracking community has gone to develop modern day filters such as Alpha Beta filters, Probabilistic Data Association filters, Interacting Multiple Model filters, and several dozen, if not hundred, variants of the aforementioned. It is most likely that these filters will find some kind of a parallelization in the near future as ever more sensors are dispersed throughout society and even more targets are monitored with these sensors. The volume of targets then becomes simply too unmanageable for a serial algorithm and more focus is placed

on parallel ones. Yet, before the parallel algorithms can be utilized they have to be derived. It is the derivation of these parallel algorithms which is the focus of this thesis. However, it should be made clear that it would be impossible to formulate a parallelization for every filter found in the literature, and so the goal here is to direct the attention onto one filter in particular, the Kalman filter.

Acknowledgements

I would like to thank my supervisor Dr. T. Kirubarajan, whose guidance and insight made this possible.

I would also like to thank Dr. T. N. Davidson, Dr. J. P. Reilly, Dr. N. Nikolova, Dr. A. Patricu, Dr. A. Emadi, Dr. R. Tharmarasa and all the professors at McMaster University who helped forge me into the engineer I am today.

Notation and abbreviations

AIS: Automatic Identification System

ANSI: American National Standards Institute

API: Application Programming Interface

BLAS: Basic Linear Algebra Subroutines

CPU: Central Processing Unit

CUDA: Compute Unified Device Architecture

FLOO: Floating-point Operation

G: Force of Gravity

GHz: Giga Hertz

GPU: Graphical Processing Unit

IDE: Integrated Design Environment

IMM: Interacting Multiple Model

JPDA: Joint Probabilistic Data Association

MB: Megabytes

MHT: Multiple Hypothesis Tracker

MP: Multiprocessors

OpenGL: Open Graphics Library

PDA: Probabilistic Data Association

RAM: Random Access Memory

SDK: Software Development Kit

Contents

List of Figures

Chapter 1

Introduction

With the advent of parallel processors and graphical processing units (GPUs) comes the promise of greater computing power and massive speed ups that translate into one thing; a better computer. It's hard to think of a GPU without being constantly reminded of the benefits it brings, especially now, in a time where the performance of GPUs is finally available to the public. For years graphical processing units existed solely in the exclusive domain of industrial programmers that targeted their efforts towards enhanced visual graphics. But then something happened, the power of parallel processing suddenly became available to everyone through APIs like OpenGL and CUDA. Before long engineers, scientists and even hobbyists began programming parallel code only to find out that the promise of enhanced runtimes came at a cost, that cost was complexity. This is why the development of source code for parallel processors has not been as progressive as some might of imagined.

Due to this complexity many programmers take the extra time to ask themselves if it's worth while writing parallel code for a particular application, because they know that once they decide to commit to developing code on a GPU, they'll have to do it

right. This means they will be burdened with the inevitable task of optimizing the source code. That may sound like a simple matter of using threads to enforce brute computations, after all, the more threads performing operations in one instruction cycle can only equate to more computations being completed in a shorter period of time. The problem is that this is far easier said than done. If one is not careful a GPU can serialize code, cause bottlenecks during data transfers to and from the device, and ultimately impeded performance instead of enhancing it. This is all possible due to the sophisticated dynamics, both in the hardware and software, of a graphical processing unit. So the only way to exploit the advantages offered by a GPU is to be clever with the computer architecture, software and the math. This is the dividing line between an optimized performance and a naive one. Surely, understanding the architecture of a GPU can only aid in its enhanced functionality, but coupling that with better mathematical algorithms will undoubtedly become the status quo. Yet, like any standard, there must be absolute scientific proof as to its validity, which, fortunately, is already the case. There are mathematical tools, some of which are hundreds of years old, that can aid any programmer in deriving clever parallel algorithms.

1.1 The Tracking Problem

The science of tracking is simple enough to describe in a single sentence; monitor the movements of a given target or a number of targets. The methodology, however, is not so easily implemented and the reasoning is vast and complicated. Firstly, differing targets behave differently. Following the movements of an oil tanker that requires a large turning radius is not the same as pursuing an unmanned aerial vehicle capable of making turns at 20-Gs [14]. Secondly, targets are commonly found in a veritable

cluster of noise and confusion that can be broken down into two main categories; the first is due to the physical limitations of the sensors that pick up noise along with the measurements, while the second is due to numerous other targets in the vicinity. The latter of these two problems calls into play the need for some kind of data association, something that has been extensively researched in various articles [26] [38] [3] [4]. Thirdly, the sensors themselves exploit many different mediums of measurements such as radar, optics, and Doppler to name a few. Couple these diverse mediums with numerous sensors operating simultaneously and the field of data fusion quickly erupts into frenzied endeavour [6] [22] [23] [5].

Yet, there is one common attribute that all of the aforementioned articles and books share and that is regardless as to what the specific tracking application is, there must be some kind of a filter [1] [9] [2] [21]. It is these filters that constitute the core of tracking and due to the probabilistic/stochastic [32] nature of these filters, the computations can become quite cumbersome, even intractable at times. So if intensive calculations need to be performed, often times on multiple targets, then the obvious solution is the use of parallel processing.

Accepting the fact that GPUs will have to be applied inevitably leads to the harsh truth that every filter must now be paralellized. There is no quick fix to this problem, the only solution is hard work. A parallel algorithm must be derived for each and every filter in turn and then implemented in software, which in itself is a time consuming task prone to bugs. Nonetheless, advances in technology dictate the necessity of such an endeavour and industry demands it, so the process has to start somewhere. This leads to the heart of this thesis, which is the starting point for the most basic of filters and the techniques that will forge them into parallel algorithms.

The good news is that selecting the first filter to parallelize is not too difficult a task since there is one in particular that stands out, one that has endured for over 50 years and is still in widespread use today; the Kalman Filter [18].

1.2 Parallel Computations

The Kalman filter is widely used in Tracking and Data Fusion and has established itself as an indispensable tool therein. Due to its broad demand, it seems reasonable that the parallelization of tracking filters begin with this highly utilized filter. This, however, draws into question the feasibility of deriving a parallel algorithm from a recursive one. The short answer is that there may be a way to parallelize the Kalman filter despite the fact that its recursive nature suggests a serial execution, but that does not mean it is a worthwhile effort. The truth is that most sensors sweep a large area and they do so at a sampling rate that is quite slow from the perspective of a computer. This means that if a single target was to be tracked, its current position, velocity and acceleration measurements would arrive long after the computer has completed the calculations from the previous measurements, amounting to a considerable waste of time for a GPU. A more realistic approach is to perform the calculations on multiple targets simultaneously. Interestingly enough, this is exactly what a GPU was meant to do, perform the same computation on differing data sets, which is not to be confused with performing differing calculations on multiple measurements. This implies that the standard way of accomplishing matrix/vector calculations using serialized software loops that perform element by element operations must be abandoned for a more strategic approach. It is this strategic plan of attack that is the basis of chapter 2, wherein the conventional methodology for executing computations between

matrices and vectors is replaced with a superior implementation that boasts the use of block multiplication [15].

Obviously, performing block operations is nothing novel, since that is the entire working basis of a GPU. But as was hinted to earlier, if one blindly executes matrix and vector computations without being conscious of the intended application, the results may prove unfavourable. So the idea is to find the best combination of GPU resources and matrix/vector dimensions, which will have to be customized for the Kalman filter. This in turn has the added benefit of establishing a methodology that can be used on the development of filters in the future.

1.3 Parallel Algorithms

Using block operations for matrix manipulation is an appropriate first step, but it is not the ultimate goal. The real prize when developing source code on a GPU is to derive a parallel algorithm. If this idea seems too vague or foreign, rest assured that it will be explained thoroughly in chapter 3. Until then, a simple analogy should help to satiate one's curiosity. Think of an ordered set of numbers, and then determine the best way to find a single value in that ordered set. The naive way would be to search through the set one at a time, starting from the beginning, while the smart thing to do would be to perform a binary search. Both operations can be performed on a CPU, and both will find the number they seek, but one is impressively faster, and all just by exploiting the fact that the set is ordered. So it becomes clear that there can be a smarter way of doing things if one is aware of all the facts.

It just so happens, that there are certain aspects of a GPU that allow an algorithm to perform far better than it normally would. This, however, brings with it certain

challenges that cannot be overcome with customary mathematical tools, instead a fresh perspective is needed and is readily available in the prefix sums algorithm [10] [13]. The prefix sums is an elegant, yet simplistic approach to paralellizing serial algorithms, and is even capable of turning a recursive algorithm into a parallel one. Its basis forms the entirety of the third chapter where a detailed dissection of its core functionality is analysed along with an in depth proof of how it can be applied to a wide range of mathematical models.

This brings forth the novelty of this thesis, which is to derive a parallel algorithm for the retrodiction of the Kalman filter that can be applied through the prefix sums operation. The derivation uses rather unfamiliar math tools aimed at turning recursive code into parallel code, an effort that marks the beginning of what will surely be a push towards the eventual parallelization of all tracking algorithms found in both academia and industry.

1.4 Publications

T. Kirubarajan, S. Tager. "GPU-Specific Kalman Filtering and Retrodiction for Large-Scale Target Tracking", To be submitted to IEEE Transactions on Aerospace and Electronic Systems, July 2013.

Chapter 2

The Kalman Filter

A formal understanding of the Kalman filter [18] is the only way to wrest an efficient mathematical algorithm that can be reasonably implemented on a GPU. This results in the need to step through the Kalman filter and all the components associated with it. It should be stated that there are many tutorials and examples in academic literature that explain the Kalman filter, yet the goal here is to observe the computational complexity and not to become overly involved with the actual derivation. Therefore, in order to maintain a clear and comprehensive picture without delving too deeply into the proof, the pages to follow will be referring to the tutorial outlined in [7].

Before exploring the mathematical details, however, it might be best to acquire a broad overview of the Kalman filter, its advantages and applications. In its most simplistic definition the Kalman filter is the optimal solution to the prediction of a linear Gaussian problem. No filter performs better predictions on measurements riddled with Gaussian noise and modelled as a Gaussian problem. One of the reasons for this exceptional performance is that the Kalman filter not only models the electrical noise of a signal as a Gaussian random variable but also the uncertainty in the

actual process itself. Since then the convention has been to call the electrical noise the measurement noise ω_k, while referring to the random error of the model as the process noise ν_k, where ω_k and ν_k are both sequences of zero-mean white Gaussian noise sampled at discrete time instances k.

2.1 One Iteration

The mathematical basis for the Kalman filter begins with nothing more than a state space equation of the dynamic system defined as

$$x_{k+1} = F_k x_k + G_k u_k + \nu_k \tag{2.1.1}$$

This equation is governed by the transition matrix F_k, which models the evolution of the states, the input matrix G_k that describes the constraints placed upon the input variables and the vector of independently distributed zero-mean white Gaussian noise ν_k, all of which are dependant on the sampling time k. For conciseness it should be mentioned that ν_k is a set of independent and identically distributed random variables defined by the following mean, variance and covariance, respectively

$$\begin{aligned} &E[\nu_k] = 0 && \text{for all } k \\ &E[\nu_k \nu_l^T] = 0 && \text{for all } k \neq l \\ &E[\nu_k \nu_k^T] = cov(\nu_k, \nu_k^T) = Q_k \end{aligned}$$

where Q_k is the covariance matrix for the process noise and the superscript T stands for the transpose operation.

After having defined the state space equation, the output of the state space model can be defined as

$$z_k = H_k x_k + \omega_k \tag{2.1.3}$$

The key to this equation is the output matrix H_k, also known as the measurement matrix, and the measurement noise ω_k, which like the process noise is independent and identically distributed zero-mean white Gaussian noise satisfying the following expectations

$$\begin{aligned} &E[\omega_k] = 0 && \text{for all } k \\ &E[\omega_k \omega_k^T] = 0 && \text{for all } k \neq l \\ &E[\omega_k \omega_k^T] = cov(\omega_k, \omega_k^T) = R_k \end{aligned}$$

There are now two covariance matrices; Q_k for the process noise and R_k for the measurement noise, both of which add to the uncertainty in the output of any linear Gaussian system.

The Kalman filter begins with the state estimation $x_{k|k}$ that is used to predict the successive state $x_{k+1|k}$, where the notation $x_{k+1|k}$ is interpreted as the value of x at time $k+1$ given the value of x at time k (Its meaning and notation are identical to that of a conditional expectation). So the predicted state is calculated using the state space matrix

$$x_{k+1|k} = F_k x_{k|k} \tag{2.1.5}$$

This predicted state is then substituted into equation (2.1.3) in order to estimate the

value of the next measurement

$$z_{k+1|k} = H_{k+1} x_{k+1|k} \tag{2.1.6}$$

The error in the estimated measurement can now be computed

$$\tilde{z}_{k+1|k} = z_k - z_{k+1|k} \tag{2.1.7}$$

At this point no further computations can be made on the measurements or the states until both the innovation covariance and filter gain matrices are computed. Both of these matrices rely on the state estimation covariance matrix which is calculated as follows

$$P_{k+1|k} = F_k P_{k|k} F_k^T + Q_k \tag{2.1.8}$$

The predicted covariance is then used to compute the innovation covariance

$$S_{k+1} = H_{k+1} P_{k+1|k} H_k^T + R_{k+1} \tag{2.1.9}$$

With the innovation covariance accounted for the filter gain is then determined according to the following

$$W_{k+1} = P_{k+1|k} H_{k+1}^T S_{k+1}^{-1} \tag{2.1.10}$$

Now would be a good time to elaborate on an important detail; up until this point neither the measurement error $\tilde{z}_{k+1|k}$ nor the filter gain W_{k+1} have any inter dependant variables besides k. This means that equations (2.1.5) to (2.1.7) can be computed alongside and independently of equations (2.1.8) to (2.1.10). Since the goal of this paper is to seek out the optimal parallel algorithm, on might be tempted to

try and calculate $\tilde{z}_{k+1|k}$ and W_{k+1} in parallel. This, however, is not possible with today's GPUs due to the fact that all the threads on the GPU perform the same operation only on differing values, and the operations involved in deriving $\tilde{z}_{k+1|k}$ are quite different from those required to obtain W_{k+1}.

The final steps of the Kalman filter involve updating the state estimations and the corresponding state estimation covariance matrix.

$$\hat{x}_{k+1|k+1} = x_{k+1|k} + W_{k+1}\tilde{z}_{k+1|k} \tag{2.1.11}$$

$$P_{k+1|k+1} = P_{k+1|k} + W_{k+1}S_{k+1}W_{k+1}^T \tag{2.1.12}$$

The updated state and covariance estimations are then fed back into equations (2.1.5) and (2.1.8), respectively and the whole procedure is repeated.

$$x_{k|k} = \hat{x}_{k+1|k+1} \tag{2.1.13}$$

$$P_{k|k} = P_{k+1|k+1} \tag{2.1.14}$$

This leaves one unanswered question; if (2.1.5) and (2.1.8) require estimates to perform the calculations, how does one start the iteration at time zero? The standard practice is to assume an initial state and covariance estimate and allow the Kalman filter to converge to the optimal estimate in a few iterations. However, caution should be applied with this approach since an initial estimate far from the actual one may cause the filter to diverge.

2.1.1 Computational Cost

The previous section gave a complete diagnostic of the computations involved in the Kalman filter. Reviewing these in turn shows that one iteration will cost 8 matrix and 3 matrix-vector multiplications, 3 matrix and 2 vector additions, 4 matrix transpose operations and a single matrix inversion. It is important to note that not all matrices and vectors contain the same dimensions. A typical example of this would be a two dimensional problem consisting of x and y coordinates, wherein each dimension has a position, velocity and acceleration state. This type of problem would render the vector $x_{k|k}$ of length 6×1, while $z_{k+1|k}$ would contain only 2×1 elements. So, in order to accurately compute the runtime [13] it is necessary to set n_s equal to the number of states and n_d the number of dimensions, where typically $n_d \leq n_s$. Then by knowing the dimension of $x_{k|k}$ and $z_{k+1|k}$ one can deduce the size of all the matrices, which leads to runtimes detailed in Table 2.1, where it has been assumed that S_{k+1}^{-1} has a runtime of $O(n_d^3)$.

No.	Operation	Variables	Runtime
1	Multiplication	$F_k x_{k\|k}$	$O((n_s n_d)^2)$
2	Multiplication	$H_{k+1} x_{k+1\|k}$	$O(n_s n_d^2)$
3	Subtraction	$z_k - z_{k+1\|k}$	$O(n_d)$
4	Multiplication	$F_k P_{k\|k}$	$O((n_s n_d)^3)$
5	Transpose	F_k^T	$O((n_s n_d)^2)$
6	Multiplication	$F_k P_{k\|k} F_k^T$	$O((n_s n_d)^3)$
7	Addition	$F_k P_{k\|k} F_k^T + Q_k$	$O((n_s n_d)^2)$
8	Multiplication	$H_{k+1} P_{k+1\|k}$	$O(n_s^2 n_d^3)$
9	Transpose	H_{k+1}^T	$O(n_s n_d^2)$

10	Multiplication	$H_{k+1}P_{k+1\|k}H_{k+1}^T$	$O(n_s^2 n_d^3)$
11	Addition	$H_{k+1}P_{k+1\|k}H_{k+1}^T + R_{k+1}$	$O(n_d^2)$
12	Transpose	H_{k+1}^T	$O(n_s n_d^2)$
13	Multiplication	$P_{k+1\|k}H_{k+1}^T$	$O(n_s^2 n_d^3)$
14	Inverse	S_{k+1}^{-1}	$O(n_d^3)$
15	Multiplication	$P_{k+1\|k}H_{k+1}^T S_{k+1}^{-1}$	$O(n_s n_d^3)$
16	Multiplication	$W_{k+1}\tilde{z}_{k+1\|k}$	$O(n_s n_d^2)$
17	Subtraction	$x_{k+1\|k} - W_{k+1}\tilde{z}_{k+1\|k}$	$O(n_s n_d)$
18	Multiplication	$W_{k+1}S_{k+1}$	$O(n_s n_d^3)$
19	Transpose	W_{k+1}^T	$O(n_s n_d^2)$
20	Multiplication	$W_{k+1}S_{k+1}W_{k+1}^T$	$O(n_s^2 n_d^3)$
21	Subtraction	$P_{k+1\|k} - W_{k+1}S_{k+1}W_{k+1}^T$	$O((n_s)n_d)^2)$

Table 2.1: Operational Complexity of the Kalman Filter

One issue with Table 2.1 is that the operations H_{k+1}^T and $P_{k+1|k}H_{k+1}^T$ were counted twice, although the more likely scenario is to compute these values once and then store the solutions into their own temporary matrices that can be used again at a later time. This would then result in 19 operations as opposed to 21.

So the total runtime is just the sum of those outlined in Table 2.1. Calculating the runtime may seem like a tedious task until one realizes that as the size of the data set becomes large the runtime of primary concern is the largest and all others can be ignored. A quick glance at the list above and it is clear that the largest runtime is

$O((n_s n_d)^3)$ and therefore, the complexity of the algorithm can be approximated to

$$O((n_s n_d)^3) \approx O(n^3) \tag{2.1.1.1}$$

Ultimately, (2.1.1.1) says that a single iteration of the Kalman filter contains a cubic runtime, which means that computing n iterations could become time consuming for large data sets.

2.2 The Parallel Approach

After having delved into the details of one iteration of the Kalman filter it should be apparent that with all the matrix/vector operations a GPU would come in very handy. This is due to the fact that there are numerous threads on a GPU, each of which can compute a single element within the matrix/vector operation simultaneously. That does not, however, aid in determining whether or not the prefix sums outlined in [10] can be implemented on the Kalman filter. This is an important consideration, and should be one of the first when looking to parallelize a recursive algorithm. The reasoning is that the prefix sums is a powerful mathematical tool that allows certain recursive calculations to be executed in parallel, as opposed to serially, which is the standard practice. Yet, one should be careful when considering whether or not to implement the prefix sums, because even though it may be possible, the reality is it might not be very helpful. The Kalman filter is a perfect example of this. This is due to the fact that in order to estimate the state at time $k+1$, one must first obtain the estimate at time k, which will depend on the estimate at time $k-1$, and so on. The best way to emphasize this is to explain the objective of the Kalman

filter. Simply put, one wishes to know the most likely state or states of the target at the next sampling time given the current one. If the approach in [10] were to be applied, it would require a complete set of states of length $k = 1, ..., N$ to be known, which would be useless. Suppose there are $k = 1, ..., 20$ measurements, then in order to predict the state at $k = 21$ one must first compute all of the estimates for $k = 1, ..., 20$. So computing 20 states in order to predict the 21^{st} is a waste of effort and in most practical applications of tracking this is not the case. Therefore, obtaining a batch of measurements of length $k = 1, ..., 20$ at time $k = 20$ from the same target does not prove very beneficial.

A more interesting scenario occurs when a multitude of measurements from numerous targets are observed. In this kind of a situation the Kalman filter can be run on a GPU and then multiple states can be calculated for multiple targets simultaneously. The only obstacle is that the threads and blocks of the GPU will have to be customized in a very methodical way. In order to carry out these calculations, it is imperative that one be aware of two factors; firstly, the number of threads being run on the GPU at any given time should be the maximum allowable, and secondly, these threads should be compatible with the dimension of the matrices involved in the calculations. The most evident manner of accomplishing this seems to be through the use of block matrices.

Although the approach outlined in the previous paragraph is actually quite innovative, it seems like an obvious step that would have surely been taken by someone before now. The reality is that GPUs are such a new topic in the field of computer science that there hasn't been much time to derive parallel code for everything. This author was simply fortunate enough to be in the right place at the right time and

stumbled across this observation that no one had yet realised. There is no doubt that someone would have sooner, rather than later, reached the same conclusion, and so it seems fair to say the timing was ripe for the implementation. That is not to say that no one has thought about parallelizing the Kalman filter, in fact this has been a topic of several publications [17] [11] over the last two decades. However, the publications before now focused on using the Kalman filter in multi sensor applications. For example, the idea in both [17] [11], is to use information from multiple sensors regarding a single target in a manner called "decentralizing". This amounts to collecting data from multiple sensors and computing the state estimate using the Kalman filter for each sensor, and then processing a global estimate based upon the estimates from the sensors. This will undoubtedly deliver a refined estimate, and each calculation can be performed in parallel, but multiple processors and sensors are required. The algorithm outlined in this thesis looks at multiple targets simultaneously processed on the same GPU, something that has not been attempted before now.

2.2.1 Compute Unified Device Architecture

If the goal is to exploit the use of threads, then a good start would be to review how those threads are structured. The problem with this is that there are several variants of graphical processors available in today's market, each one with their own distinct architecture. Therefore, the only practical thing to do is to focus on one, which will not undermine any of the work presented here, since the goal is to demonstrate an algorithm that is universally applicable to any GPU. The only setback in demonstrating the proposed algorithm for a specific graphics card is that if someone wanted to port the algorithm to another platform, there might have to be altercations when

allocating the threads and blocks.

In this thesis the GPU of choice will be a CUDA GeForce GTX 570 created by NVIDIA. The details of this particular brand and model are plentiful, yet since the proposed algorithm does not call for all of the peripherals of this device to be used (i.e. atomic operations, textured memory, constant memory, etc.), focus will be placed upon the main components of interest. Table 2.2 provides a good overview of the hardware that pertains to the application in question. For a more in depth look at the graphics card, the reader is referred to Appendix C.

Description	Value
Device	GeForce GTX 570
CUDA Capability	2.0
Total amount of global memory	1280 MB
Number of Multiprocessors	15
Number of CUDA Cores/MP	32
Total number of CUDA Cores	480
Total amount of shared memory per block	49152 bytes
Maximum number of threads per multiprocessor	1536
Maximum number of threads per block	1024
Maximum block size	$1024 \times 1024 \times 64$
Maximum grid size	$65535 \times 65535 \times 65535$

Table 2.2: Device Query

A good strategy when dealing with GPUs is to use as many threads as possible. This might lead one to believe that no more than 1,536 threads can be used on a

single multiprocessor according to Table 2.2. However, what is really meant is that no more than 1,536 threads can be launched simultaneously. One could have 65,000 threads allocated, but only a maximum of 1,536 of them will be working at any one given moment on any one given multiprocessor. Furthermore, if there were 65,000 threads allocated there would have to be an appropriate number of blocks allocated. Looking at Table 2.2, it can be seen that a maximum of 1,024 threads can be allocated per block. So the idea is to balance the number of threads along with the number of blocks in order to optimize the number of active threads.

2.2.2 Customizing the Threads and Blocks

To date there is no solution that guarantees optimal performance on a GPU. In fact there are reports generated daily that explore different strategies concerning the best functionality [35], [20] [31]. So it must be clear that the strategy outlined here does not claim to be the optimal one, and even if it was there is sure to be some improvement of the device hardware or software sometime in the near future that would render the optimization algorithm obsolete. Yet, there is a straightforward strength to the idea that is novel enough to make mention of.

The application herein was adopted for the explicit use of a predefined dimension and state size. Surely, it would seem intuitive to design software that can adapt to any state or dimension size, however, a problem could arise wherein a state or dimension size exceeding the maximum number of threads per block would return an erroneous solution. As such, from here on out it will be assumed that the state and dimension size are well within the bounds of the thread count outlined in Table 2.2. This constraint enables the designer to partition each block into a series of matrices

and vectors (refer to Appendix A for a complete review of matrix partitions). The advantage then becomes apparent; in place of element by element multiplication and addition, as is normally done within a serial algorithm, one may perform block multiplication and addition[1]. Extending this idea, suppose there are s states and r dimensions, then the estimated covariance matrix will be an $sr \times sr$ matrix. All that is needed now is to tile this matrix, wherein the total number of threads do not exceed those prescribed by the GPU, which in this case is 1,024. Lastly, build a grid of GPU blocks all the while ensuring that there is enough shared memory.

In order to leave no detail unattended, a practical example will be devised and tested. Suppose the system has two dimensions, x and y, and that each dimension has three states, namely position, velocity and acceleration, then the estimated covariance matrix will have dimensions 6×6. It would then seem appropriate to allocate 576 threads per GPU block, wherein there are 24 threads along the x dimension and 24 threads along the y dimension, resulting in a block matrix like the one shown below.

$$P_1 = \begin{bmatrix} P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & P_{22} & P_{23} & P_{24} \\ P_{31} & P_{32} & P_{33} & P_{34} \\ P_{41} & P_{42} & P_{43} & P_{44} \end{bmatrix} \tag{2.2.2.1}$$

where all block matrices P_{11}-P_{44} are of size 6×6.

The next thing to consider is the total amount of shared memory required. In the sections to come it will become clear why a total of 4 GPU blocks were allocated,

[1]It is important to note that block multiplication and addition does not refer to the blocks on the GPU. The nomenclature of "block matrices" is well defined in the literature while NVIDIA has opted to call a portion of their architecture "blocks". Should this cause any potential confusion then the term "block" will refer to matrices and "GPU blocks" will represent the hardware partitions on the NVIDIA GPU.

but until then it shall be assumed that this is the maximum number of GPU blocks (given each GPU block has 576 threads) that will execute successfully on the device in question. Using the estimation covariance matrix P as an example, the final size of the matrix to be computed will be 24×96, wherein there are 4 block matrices, each of size 24×24.

$$P = \begin{bmatrix} P_1 & P_2 & P_3 & P_4 \end{bmatrix} \tag{2.2.2.2}$$

Since the matrix F_k is the same size as $P_{k|k}$, then it too can be expressed in an identical block form

$$F = \begin{bmatrix} F_1 & F_2 & F_3 & F_4 \end{bmatrix} \tag{2.2.2.3}$$

2.2.3 Block Operations

The ultimate goal here is to determine how the GPU will perform the multiplication between the two matrices defined in (2.2.2.2) and (2.2.2.3). However, this is the final realization and it is first necessary to review the complexity of a single matrix multiplication. Therefore, it will simplify matters to look at a single block of size 6×6, namely F_{11} and P_{11}. Referring to (2.1.8), and concentrating on a single matrix multiplication,

$$\begin{bmatrix} f_{11} & f_{12} & f_{13} & f_{14} & f_{15} & f_{16} \\ f_{21} & f_{22} & f_{23} & f_{24} & f_{25} & f_{26} \\ f_{31} & f_{32} & f_{33} & f_{34} & f_{35} & f_{36} \\ f_{41} & f_{42} & f_{43} & f_{44} & f_{45} & f_{46} \\ f_{51} & f_{52} & f_{53} & f_{54} & f_{55} & f_{56} \\ f_{61} & f_{62} & f_{63} & f_{64} & f_{65} & f_{66} \end{bmatrix} \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} & p_{15} & p_{16} \\ p_{21} & p_{22} & p_{23} & p_{24} & p_{25} & p_{26} \\ p_{31} & p_{32} & p_{33} & p_{34} & p_{35} & p_{36} \\ p_{41} & p_{42} & p_{43} & p_{44} & p_{45} & p_{46} \\ p_{51} & p_{52} & p_{53} & p_{54} & p_{55} & p_{56} \\ p_{61} & p_{62} & p_{63} & p_{64} & p_{65} & p_{66} \end{bmatrix} \tag{2.2.3.1}$$

it becomes possible to grasp a firm understanding of how the computations are performed. The GPU will multiply the first column of $P_{k|k}$ by all six 6 rows and all 6 columns of F_k in a single iteration. This results in 36 floating-point operations (FLOPs)[1] on a CPU, and only one on a GPU.

$$\begin{bmatrix} f_{11}p_{11} + f_{12}p_{21} + f_{13}p_{31} + f_{14}p_{41} + f_{15}p_{51} + f_{16}p_{61} \\ f_{21}p_{11} + f_{22}p_{21} + f_{23}p_{31} + f_{24}p_{41} + f_{25}p_{51} + f_{26}p_{61} \\ f_{31}p_{11} + f_{32}p_{21} + f_{33}p_{31} + f_{34}p_{41} + f_{35}p_{51} + f_{36}p_{61} \\ f_{41}p_{11} + f_{42}p_{21} + f_{43}p_{31} + f_{44}p_{41} + f_{45}p_{51} + f_{46}p_{61} \\ f_{51}p_{11} + f_{52}p_{21} + f_{53}p_{31} + f_{54}p_{41} + f_{55}p_{51} + f_{56}p_{61} \\ f_{61}p_{11} + f_{62}p_{21} + f_{63}p_{31} + f_{64}p_{41} + f_{65}p_{51} + f_{66}p_{61} \end{bmatrix} \tag{2.2.3.2}$$

Each row of the resulting matrix must be summed and the answers stored in the first column of the solution matrix. The addition of these values would normally be done one at a time resulting in 5 addition operations per row for a total of 30 FLOPs. Yet, the approach used here will be adopted from the prefix sums algorithm [10]. Therefore, column 1 will be added to column 2 and the result stored in column 1, column 3 will be added to column 4 and the result stored in column 3, and column 5 will be added to column 6 with the result stored in column 5, all in one FLOP. Column 1 will then be added to column 3 and stored in column 1, which will require a single FLOP and finally column 5 will be added to column 1 and the final answer stored in column 1, which will account for the final FLOP. So instead of 30 FLOPs there will be 3.

[1]From now on a FLOP will refer to the number of floating-point operations in a single instruction cycle on a CPU, while FLOPs will be regarded as the plural of a FLOP and not to be confused with floating-point operations per second.

The next iteration will multiply the entirety of F_k by the second column of $P_{k|k}$, again in a single FLOP. Then 3 more FLOPs will be required to sum all the columns and the answer stored in the second column. This will continue until F_k has been multiplied by all 6 columns of $P_{k|k}$, which brings the total number of FLOPs to 6 multiplications and 18 additions. Therefore, the multiplication of two matrices requires a runtime of $O(n)$ for both addition and multiplication operations, assuming that the dimensions of the estimated covariance and the state space matrix are $n \times n$. Comparing this to a serial algorithm that would have a runtime of $O(n^3) + O(n^2 - n)$, and the benefits of a GPU become blatantly obvious.

Recall that the matrix multiplication above was for a single block matrix. There are, however, 16 matrices in (2.2.2.1), and the manner in which the GPU processes these is one row at a time. Using the product below to elaborate on this point,

$$F_1 P_1 = \begin{bmatrix} F_{11} & F_{12} & F_{13} & F_{14} \\ F_{21} & F_{22} & F_{23} & F_{24} \\ F_{31} & F_{32} & F_{33} & F_{34} \\ F_{41} & F_{42} & F_{43} & F_{44} \end{bmatrix} \begin{bmatrix} P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & P_{22} & P_{23} & P_{24} \\ P_{31} & P_{32} & P_{33} & P_{34} \\ P_{41} & P_{42} & P_{43} & P_{44} \end{bmatrix} \tag{2.2.3.3}$$

It is seen that each block along the first row of F_1 is multiplied by the blocks along the first column of P_1, while the second row of F_1 is multiplied by the second column

of P_1, resulting in the following matrix

$$\begin{bmatrix} F_{11}P_{11} & F_{12}P_{21} & F_{13}P_{31} & F_{14}P_{41} \\ F_{21}P_{12} & F_{22}P_{22} & F_{23}P_{32} & F_{24}P_{42} \\ F_{31}P_{13} & F_{32}P_{23} & F_{33}P_{33} & F_{34}P_{43} \\ F_{41}P_{14} & F_{42}P_{24} & F_{43}P_{34} & F_{44}P_{44} \end{bmatrix} \tag{2.2.3.4}$$

The previous calculations showed that every block multiplication will cost a runtime of $O(n)$, however, every block along the first row will be computed simultaneously, which still only costs $O(n)$. This must be performed for each row in turn, and since there are only 4 rows, the cost is still only $O(n)$. A good way to visualize this is to realize that the entire matrix is 24×24 and all 24 threads along the first six rows of F_1 are multiplied with the 24 threads along the first column of P_1. A CPU will require $(O(n^3) + O(n^2 - n))$, yet, the advantages of the GPU don't end there. Recall that there are 4 GPU blocks, as outlined below

$$FP = \begin{bmatrix} F_1P_1 & F_2P_2 & F_3P_3 & F_4P_4 \end{bmatrix} \tag{2.2.3.5}$$

and each GPU block is executed in parallel. So in fact, all of the aforementioned computations are executed simultaneously across all 4 GPU blocks, which means there are a total of 96 threads across 6 rows of F multiplied by the 1^{st}, 7^{th}, 13^{th} and 19^{th} columns of P all at once. Furthermore, as was seen above, the runtime for F_1P_1 is $O(n)$, and all 4 GPU blocks are launched simultaneously and executed in parallel, so whether 1 block is used or 4, the runtime remains $O(n)$. Refer to Table 2.3 for the runtimes pertaining to the block operations explained above.

After all of the math is done and said it is no surprise that the CPU has a cubic

runtime, something that is well established [13]. However, the GPU does present a welcomed advantage with a runtime that is nearly linear, but at what cost?

Operation	GPU	CPU
Matrix Multiplication	$O(n)$	$O(n^3) + O(n^2 - n)$
Block Multiplication	$O(n)$	$(O(n^3) + O(n^2 - n))$
Grid Multiplication[1]	$O(n)$	$(O(n^3) + O(n^2 - n))$

Table 2.3: Run Time Operations

Before moving onto the next section, there is something of importance that needs to be addressed. The last two sections outlined a strategy for implementing matrix/vector computations using the CUDA runtime library. This, however, is by no means the only tool available to a designer, another very popular API is OpenGL [19]. All of the block operations detailed in this section could easily be ported to OpenGL, where the syntax will undoubtedly vary, and there may have to be a few more minor changes, but the algorithm is disposable on any API that utilizes a graphical processing platform.

2.2.4 The Source Code

This section steps through the relevant parts of code that perform the operations discussed in section 2.2.3. The first thing to look at will be memory allocation. A quick note to the reader, although in Table 2.1 it was determined that there are 21 matrix/vector operations, matrix-transpose multiplication can easily be performed on a GPU in one step. This means that the operations $P_{k|k}F_k^T$, $P_{k+1|k}H_{k+1}^T$ and

[1] Where a grid is comprised of all 4 GPU blocks

$S_{k+1}W_{k+1}^T$ can each be performed without having to take the transpose explicitly. Normally, the rows of the multiplicand matrix are multiplied by the columns of the multiplier matrix, but if the rows of the multiplicand are instead multiplied by the rows of the multiplier, then this is the equivalent of multiplying the multiplicand by the transpose of the multiplier. Fortunately, this is a simple undertaking for GPU, since the threads along the columns of the multiplier can be swapped with those along the rows. This then amounts to 17 matrix operations (since $P_{k+1|k}H_{k+1}^T$ is performed once and stored in memory for later use a second time) as opposed to 21.

There are several variants of memory available to the user on a GPU, but only two of those are of great concern here; global memory and shared memory. These two types of memory vary in their size and speed. Global memory is more plentiful and accessibly interchangeable to every grid, block and thread. This means that any thread residing in any block located on any grid can be added, subtracted, multiplied or divided by any other thread. Threads using shared memory, however, can only perform computations with other threads that exist in the same block. This limits the accessibility of the threads, but the access speed of shared memory is several times faster than that of global memory. The speed of shared memory is due to its location directly on the multiprocessor, while global memory is located on the GPU but not on the multiprocessor [29].

The diversity of the hardware forces a programmer to become familiar with the specifics of a GPU if the desired speed up is to be achieved. Whether one should use shared or global memory is dependant on the intended application. One might think that since shared memory is faster than global memory, the smart thing to do is to use it. The problem is, however, that data is transferred from the host to

the device's global memory by default, and if one wishes to use shared memory then an additional transfer out of global memory and into shared memory must be made before operations can begin. So the only time it makes sense to use shared memory is if there are a sufficient number of computations that warrant this data transfer. If the GPU is being used to simply add the contents of two vectors, global memory might suffice. The need for shared memory is usually justified when there are a large number of calculations that are more advanced than simple addition or multiplication, which means that the GPU will be spending a large percentage of its time performing these calculations and very little time transferring data to and from global memory.

In the case of the Kalman filter and its numerous matrix/vector operations, the use of shared memory does make sense. As such, every matrix/vector must be allocated enough shared memory to store 64 matrices/vectors appropriately sized for 3 state space variables and 2 dimensions. This means that the state estimation vector $\hat{x}_{k|k}$ should be allocated 384 array elements, while the process noise Q_k should be given 2,304 array elements, both of type `float`. The programmer should always take the time to do a quick calculation in order to ensure that there is an appropriate amount of shared memory allocated. If too much memory is allocated then the program will crash, not enough and the program will be too slow. The amount of shared memory available can be confirmed using Table 2.2, where it explicitly states that the maximum amount of shared memory per block is 49,152 bytes.

Since the kernel will be taking advantage of shared memory, before any operations can be performed the data must be transferred from global memory. The device works quite simply; when data is transferred from the host to the device it is automatically stored into global memory. From there the programmer must decide where to send

it and what to do with it. In this case the data will be transferred from global memory to shared memory from where the computations will be executed. Once those calculations are done the data must be transferred from shared memory back into global memory where it can be sent back to the host. So, the moral of this story is to ensure that there are enough computations that justify the use of shared memory, since there will be two extra data transfers, to and from shared memory.

The next thing to consider is how the matrix/vector multiplications are being executed. Listing 2.1 looks at the complete matrix multiplication of F_k and $P_{k|k}$

Listing 2.1: Matrix Multiplication

```
for(j=0; j<4; j++)
{
  for(i=0; i<6; i++)
  {
    if(row < 6)
    {
      B[y][col] = F[y+6*j][col]*P_k_k[x][6*j+i+24*bx];

      if((x%2) == 0)
      {
        B[y][col] += B[y][col+1];

        if((x%6) == 0)
        {
          B[y][col] += B[y][col+2];
```

```
            B[y][col] += B[y][col+4];
          }
        }

        if((x%6) == 0)
        {
          P_k_k[y+6*j][col+i] = B[y][col];
        }
      }
      __syncthreads();
    }
}
```

As was explained in section 2.2.3, the first `for` loop in Listing 2.1 consists of the 4 iterations that perform block multiplications on each row of (2.2.2.1). The second `for` loop computes the multiplication of F_k with each column of $P_{k|k}$, hence the need for 6 iterations, one for each column of $P_{k|k}$. The 3 addition operations are performed to sum all the columns after F_k has been multiplied by a single column of $P_{k|k}$, therefore, these additions are performed 6 times, for a total of 18 FLOPs. Then everything is stored back into $P_{k|k}$ for future use.

The matter concerning the inverse computation of the innovation covariance in (2.1.10) can be solved quite easily by solving for the determinant [36]. Since this is a 2 dimensional problem, the size of S_k is 2×2 and therefore S_{k+1}^{-1} can be calculated

using the following formula [36]

$$S^{-1} = \frac{1}{s_{11}s_{22} - s_{12}s_{21}} \begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix} \tag{2.2.4.1}$$

Once all the computations are complete the last order of business is to transfer the result back into global memory where the host can retrieve the data.

2.2.5 The Price of Parallel

The first thing every scientist and engineer learns is that if its too good to be true, then that's exactly what it is. If graphical processors provide such a blatant advantage, then why isn't everything done in this manner? One obvious answer is that the technology is relatively new and there simply hasn't been enough time to convert all of the serial algorithms into parallel ones. Although this is relatively correct, a more realistic answer is that it may not be worthwhile to derive a parallel algorithm due to its complexity, which is the price one pays for writing software on a GPU. If the GPU provides an exponential speed up then it does so at a cost of exponential code complexity due to difficult indexing and a heightened hardware diversity. These two points emphasize what is being discussed here, so it would seem wise to explore both.

The idea that an advanced architectural layout would impede/improve performance was discussed in the previous section where data was first transferred from global to shared memory. This is not so different from booting up a computer during which time the CPU transfers data from the hard drive to RAM. After that transfer is complete the computer uses RAM exclusively for the lifetime of any program without the need to move the data to a different memory register for improved functionality.

So in essence, global memory can be thought of as a hard drive, and shared memory is the equivalent of RAM. The difference, however, begins with the number of threads and the size of the blocks that can limit or extend performance. The fact that there are varying memory registers with differing execution properties is the responsibility of the programmer to optimize. Yet, before anyone can exploit the use of something like shared memory they have to be aware of its existence, hence the need for a refined understanding of the hardware architecture of a parallel processor and all of the peripherals there within.

The second issue worth mentioning is concerning the indexing of the arrays within the GPU that are emphasized in Listing 2.1. A close inspection is enough to convince the reader that there are up to 6 different indexing variables in a single iteration, and that is for one matrix multiplication. Recall that the implementation outlined here requires 17 operations on matrices and vectors of varying dimension, which means that the indexing of most matrix/vector operation will be unique. Be advised that debugging this is no small feat. It could take a day or two to devise a strategy, much like partitioning everything into block matrices, while writing the actual software might take a month or two.

When delving into the details of a CUDA device one quickly becomes overwhelmed by the peripherals offered. Having to deal with threads located in blocks embedded in grids that can be optimized through shared memory, textured memory, atomic operations and asynchronous data transfers exploited through the use of streams [30] is enough to dissuade any programmer from taking the time to write parallel programs. Add to this the constant responsibility of being ever aware of the number of threads and blocks and how they behave according to the rules governed by the

warp sizes thus affecting the occupancy, while taking care to ensure that portions of the code do not become serialized causing issues with concurrency [28] and it becomes clear why most programs will not see a parallel implementation. The only time a serial algorithm should be converted into a parallel one is if the ends justify the means, which is the case for tracking and data fusion.

The specific example from sections 2.2.2 and 2.2.3 showed that the block computations were customized for a GPU that had exactly 24×24 threads and 4 blocks allocated. These values were hard coded into the program and any alteration would render the algorithm useless. If one desired to implement the code on a 3 dimensional problem, wherein each dimension had 3 states, one could no longer use 16 block matrices since the size of each matrix would be 9×9 and that would demand 1,296 threads, which exceeds the number of threads per GPU block and may cause serialization. The alternative would be to implement 9 block matrices, each of size 9×9, thus utilizing a total of 729 threads. This, however, would alter the indexing of the arrays within the kernel and result in an erroneous output. The only way around this is to rewrite the kernel in order to handle the new indexing requirement.

So being aware of all the variables involved in the execution of a program that enables one to write optimal code can quickly become cumbersome, which is why no claims have been made that the source code written here is the best possible code. The parallel algorithm outlined in the aforementioned pages is definitely linear and will theoretically outperform any serial algorithm, but if one is not careful, the source code may not.

2.2.6 Simulation Results: CPU vs GPU

The following simulation was split into two distinct trial runs. The first simulation was performed on an Intel Core i7 870 @ 2.93 GHz using an ANSI C compiler. All of the code was written by this author in the C programming language so as to guarantee that the algorithm being executed was that which is outlined in section 2.1. The second component of the simulation was conducted using an NVIDIA GeForce GTX 570 graphics card and the CUDA runtime library. It should be noted that unlike the experiment run in [12], CUDA BLAS was not used here. Although there are a total of 21 matrix computations according to Table 2.1, the authors of [12] chose to partition the Kalman filter into 5 distinct computations. Whether or not they transferred the 5 partitioned sets of matrices and vectors to the GPU one at a time or all at once is unclear, but two facts do stand out; in order to perform the 21 matrix/vector calculations using CUDA BLAS one would either have to call each matrix function from the host, or make the call from within the kernel. Calling 21 matrix/vector operations from the host would require 21 data transfers from the CPU to the GPU and then another 21 back to the CPU, which would adversely affect the throughput. Calling the functions from within the device severely limits the amount of data that can be processed in a single kernel call. The reason for this is simple; there is a finite amount of available memory on the GPU and every function call from the kernel to another kernel requires memory to be allocated on the GPU. This is equivalent to pushing a function call and all the necessary memory onto a stack, and the stack is only so large. In fact, referring to Table 2.2 it can be seen that the total amount of global memory is 1,280 Mbytes, and worse yet, if shared memory is used then there are only 49,152 bytes. Furthermore, CUDA BLAS could not implement the strategy

outlined in section 2.2.3 because it performs block operations in a manner that is different from the customized block operations derived in this thesis. This is most likely why the authors of [12] chose to vary the number of dimensions between 1,000 to 7,000 and the number of states between 250 to 3,500. A matrix of this size is exactly what CUDA BLAS was meant for, but for the purposes of tracking this is unrealistic, this author has never heard of a practical situation wherein there were 250 states, much less 3,500. So instead, this thesis has opted to transfer the entire data set to the GPU once, allow the computations to be performed on the GPU, and then transfer the results back. This may not be the best way to transfer the data, since there is some indication in the upcoming sections that breaking up the data into streams and transferring each stream separately is the preferred method. Yet, performing asynchronous data transfers is outside the scope of this thesis, and will not be resolved here.

The sole task that remains is to report the results of the source code. This consists of nothing more than running the kernel on data sets of varying size and recording the results. However, before doing that it would be best to elaborate on one issue. The algorithm up until now has focused on a predefined data set of 64 measurements. Recall that the estimation covariance matrix in (2.2.2.2) was in block form with dimensions 24×96, wherein there were 64 block matrices of size 6×6. This means that the code will only work on a sample set of size 64. To workaround this limit, all the data transferred from the host is stored in global memory and from there 64 measurements at a time are transferred to the shared memory where they are processed by the kernel. Once finished, the kernel transfers the results back into global memory and the next chunk of 64 samples is swapped into shared memory for

processing. This continues until everything in global memory has been dealt with. Of course, there arises an issue with any data set that is not a multiple of 64, but that can easily be resolved by padding the data with zeros.

In order to make effective use of the GPU, one must be able to reliably gauge its performance through the use of various metrics. The CUDA Toolkit provides several ways of doing this, but by far the most useful are the timers in its libraries. At the time of this publication, the CUDA Toolkit 5.0 was the most recent API release, offering timers capable of measuring, not only the elapsed time of the GPU but that of the CPU as well. As such, CUDA SDK timers were utilized to compare the duration of code execution between the host and device across a wide range of data set sizes and the following results were recorded and are displayed in Table 2.4.

Data Size	CPU Time (s)	GPU Time (s)
64	0.005824	0.000734
128	0.034944	0.001281
256	0.009531	0.00213
512	0.011083	0.003686
1024	1.507853	0.007249
2048	1.046539	0.013867
4096	0.33041	0.027721
8192	0.419372	0.054891
16384	0.335895	0.109429
32768	2.47752	0.218099
65536	4.957672	0.434744
131072	9.861555	0.868823

262144	20.002492	1.737119

Table 2.4: Prediction Simulation Results

These results are not surprising, and furthermore, it is comforting to see that the GPU behaves exactly as predicted; in a linear fashion. Doubling the data size doubles the execution time regardless as to the number of measurements being processed. And irrespective as to how the CPU behaves for smaller data set sizes, it operates as one would expect when encumbered with larger data sets, save one feature; it is linear, and not cubic. This is the result of a small data set. If the data set were increased it should demonstrate cubic behaviour. How the host performs relative to the device can be seen in Figure 2.1, where larger data sets force the CPU runtime to diverge quickly away from the GPU.

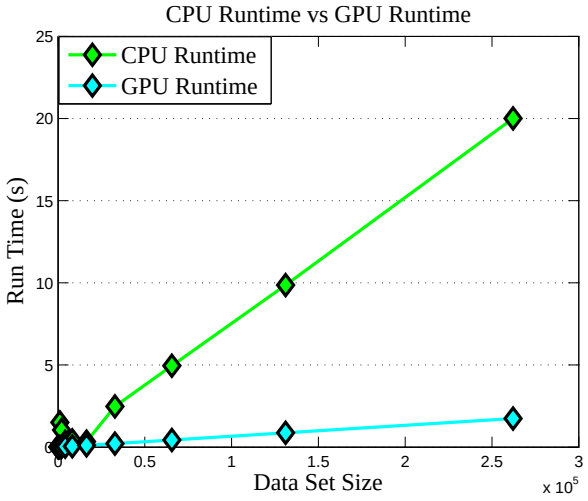

Figure 2.1: Prediction Runtime Comparison

One of the more interesting observations drawn from Table 2.4 is in regard to the execution times of the CPU for the data sizes up to and including 16,384; notice

how incoherently they vary. The data with 16,384 measurements takes less time to execute than one which is 8 times smaller. In fact, there seems to be some discrepancy concerning the data sets of size 1,024 and 2,048. Why these particular sizes cause exacerbated runtimes is a good question indeed. At one point it seemed implausible, and so in place of using the timers offered in CUDA 5.0, the event timers offered in the CUDA 4.0 were implemented. The event timers produced the same results as the SDK timers.

When reviewing Figure 2.1, it is difficult to grasp exactly how irregular the CPU behaves due to the clustering of data set sizes for small values. Therefore, the same data set from Table 2.4 was plotted logarithmically in Figure 2.2, where it becomes much more apparent that there is something peculiar regarding the behaviour of the CPU. The problem is not so much that the behaviour is varied, but that it seems inexplicable. There are several theories, such as the C compiler is performing optimization routines that run into problems with certain data sizes. Or perhaps the manner in which data is pushed and popped onto and off of the stack encounters problematic scenarios that the compiler is unaccustomed to dealing with. It could be a bug within the integrated design environment (IDE) or the operating system itself. One might wonder why the GPU shows absolutely no signs of any bad conduct that the CPU displays. The simple answer is that the GPU executes the source code within the kernel, and nothing else. Besides the speed at which the data is transferred from the host to the device, the programmer is in complete control of everything that happens on the device hardware. In contrast, the user has limited control over the operating system kernel, which has complete control of the CPU, and it is this dynamic that is the most likely culprit of this unexplained behaviour.

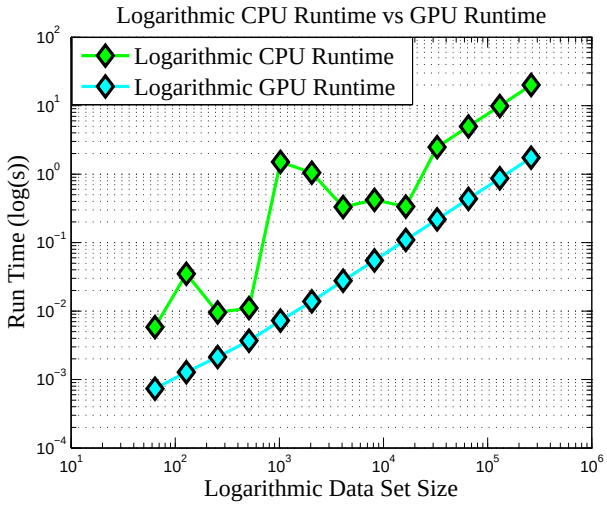

Figure 2.2: Logarithmic Runtime Comparison

There is one more issue that every software developer working with a GPU should address, and that is the accuracy. Truncation error can be a serious issue if the programmer is not conscious of it, since this can alter the results of a simulation quite substantially. The GPU used throughout this chapter supports the use of 32 bit floating point data types (`float`), which greatly simplified the comparison of results to that of the CPU, since it too is equipped with the ability to use arrays of type `float`. Both the GPU and the CPU were programmed with a data type of `float` and the results produced were identical. As such, there was no truncation or rounding off errors and therefore, the programmer is not burdened with addressing this topic, although one should be aware of it since this is an issue that may affect future development of tracking filters.

2.2.7 Simulation Results: Block Size

It was mentioned in section 2.2.2 that there is no way to predetermine the optimal performance of the GPU, due in large part to the fact that the technology is relatively new and scientists and engineers are just beginning to comprehend its limitations and advantages. There are, however, obvious steps that can be taken to improve performance at the initial design stages of the source code. One of these is to maximize the number of operational threads and blocks. Referring to Table 2.2, one can see that there are a maximum of 1,536 threads that are operational on any given multiprocessor at any given time. A good strategy would be to allocate 1,536 threads all the time, yet this is usually not possible. So the next best thing is to allocate the closest number of threads. Recall that the code being executed in this section utilizes 576 threads per block, and there are a total of 4 blocks. What if the number of blocks were varied? How would this affect the performance? Table 2.5 shows the results of running the source code while allocating 1, 2, and 4 blocks. Keep in mind that the size of the data set and the number of threads per block is identical for all 3 simulations. The reason that this is worth mentioning is due to the fact that the runtimes seem to be vastly dissimilar. There is some indication that doubling the block size halves the runtime, and so one might be inclined to keep doubling the block size, yet there is a limit to how many blocks that can be allocated before the device runs out of available shared memory.

Data Size	1 Block	2 Blocks	4 Blocks
64	0.002161	0.001049	0.000734
128	0.003859	0.002228	0.001281
256	0.0066391	0.003795	0.00213

512	0.0112965	0.006595	0.003686
1024	0.025697	0.013189	0.007249
2048	0.050348	0.026149	0.013867
4096	0.099976	0.051403	0.027721
8192	0.198816	0.102787	0.054891
16384	0.397458	0.204695	0.109429
32768	0.794539	0.408696	0.218099
65536	1.587160	0.815729	0.434744

Table 2.5: Block Size Runtimes

Based on the thread size detailed in section 2.2.2, the maximum number of blocks that can be allocated is 4. This limiting factor is due to the shared memory allocated within the kernel, which is 46,080 bytes. Notice how Table 2.2 states that the total amount of shared memory is 49,152 bytes, this value is exceeded for any block size greater than 4. A quick glance at Figure 2.3 should convince the reader how important selecting the proper block size is, while an in depth look into Table 2.2 versus Table 2.5 should assure the reader that no matter what the block size is the GPU outperforms the CPU at every turn. However, a word of caution, although using any number of blocks improved performance, this may not be the case for all algorithms. In fact, it is possible that the manner in which the threads were used in this particular code is the dominating factor, since the number of threads were never altered the GPU was never slower than the CPU. That is not to say altering the block size for another algorithm or a differing thread count will produce such favorable results.

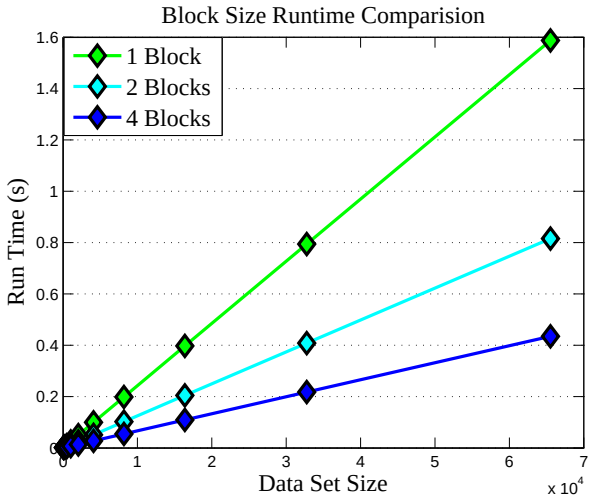

Figure 2.3: Block Size Comparison

2.2.8 Future Considerations

As was mentioned in section 2.2.5, a program written for a GPU comes with the burden of having a multitude of architectural considerations that can impede or improve the performance of one's code. A solid understanding of this computer structure will only prove to empower the programmer in compiling better code. Yet, for most this is something that will require years of experience and doesn't serve one very well in the short run. For this reason, NVIDIA has developed a tool that helps facilitate the development of optimal code, it is called Visual Profiler. Once the source code has been compiled into an executable, it can be analysed by Visual Profiler. This software tool provides invaluable information concerning performance metrics such as concurrency, warp divergence and throughput, to name a few. This knowledge is indispensable when one is engaged in an iterative design process.

After having run the source code from section 2.2.4 in Visual Profiler, several promising results were revealed that highlight a direction of potential improvement,

and have been listed below.

1. Low memory copy and compute overlap
    - The percentage of time when memory copy is being performed in parallel with compute is low
2. Low memory copy throughput
    - The memory copies are not fully using the available host to device bandwidth
3. Low kernel concurrency
    - The percentage of time when two kernels are being executed in parallel is low
4. Low global load efficiency
    - Global memory loads may have a poor access pattern, leading to inefficient use of global memory bandwidth
5. High branch divergence overhead
    - Divergent branches are causing significant instruction issue overhead.

Items 1, 2 and 3 pertain to the same problem, that being there are no asynchronous data transfers. This is due to the fact that by default the kernel will begin executing code once all of the data is transferred from the host. It is possible, however, to command the kernel to begin operations before it receives all of the data, a feat

accomplished by the use of streams. Essentially, streams allow one to break the data up into subsections and transfer one subsection while the kernel processes another.

Item 4 is in regards to coalesced access to global memory, which is a consequence of warps being misaligned when accessing global memory [31]. The final item has to do with the fact that threads within the same block that are forced through differing paths due to `if` statements or `for` loops can become serialized, which would defeat the purpose of using a GPU.

Although the items in the above list are informative, they do not represent the primary goal, which is to derive a parallel algorithm. The reason that this is so important is that however impressive the threads and blocks on a GPU perform, they are still doing calculations on a serial algorithm and that will always limit the potential of the GPU. A designer should look past the parallelization of arithmetic operations to the parallelization of entire algorithms. That is the first step in an iterative design process intended on finding the optimal solution, and hence the optimal performance. So, the issues addressed in this section, as interesting as they are, were meant to aid in improving one's software skills, and do very little to help develop solid algorithm design skills. However, optimal code requires a good algorithm and equally effective software, so this section helps remind one of what future pitfalls to avoid and where potential avenues of improvement may reside, thus paving the way for future refinements that will not be dealt with here.

Chapter 3

Retrodiction

Retrodiction is a refined estimate of the previous prediction made by the Kalman filter, which in laymen terms means it is looking back in time, not ahead. This may seem futile, but in fact has several practical applications, mostly in image processing and tracking. In tracking, it enables the tracker to acquire a polished version of the trajectory laid out by the Kalman filter. One of the best ways to demonstrate the difference between the Kalman filter and retrodiction is graphically. To be clear, the black line in Figure 3.1 is the true trajectory of the target and the red line is the measurement that has been contaminated with white Gaussian noise, while the blue and green lines are the filters in question. Notice how the retrodiction of the Kalman filter is not nearly as jagged as the Kalman filter itself, it is much smoother, hence the term "Smoothing". It gives a far better estimate of the true trajectory of the target, despite the measurements that have been riddled with noise.

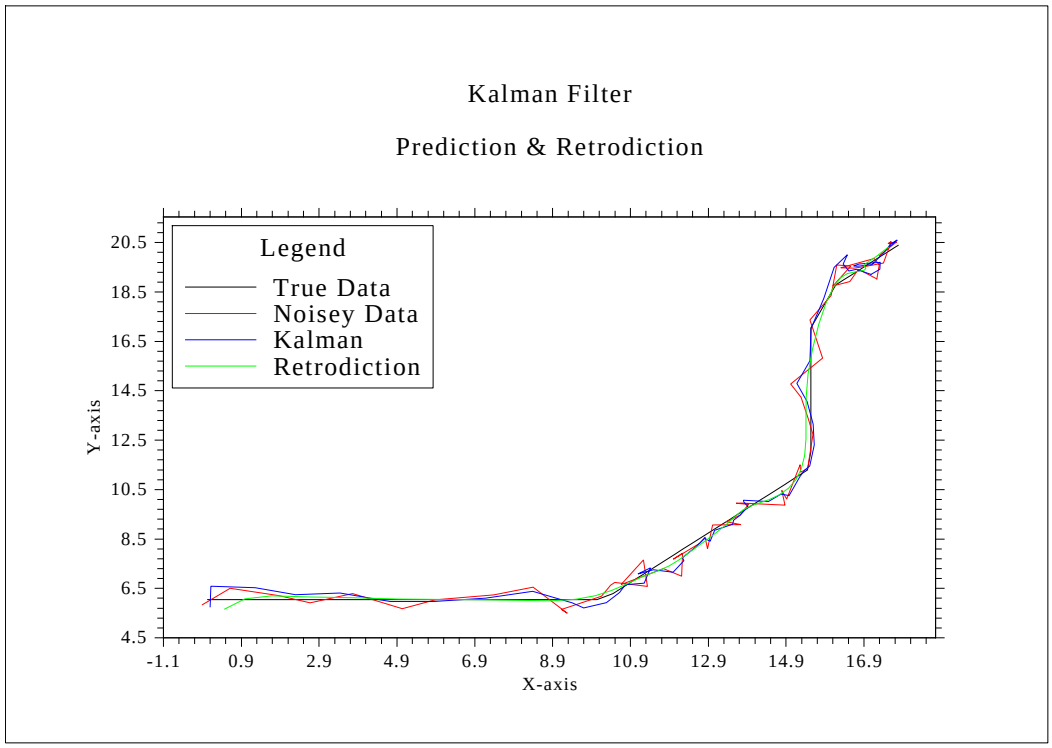

Figure 3.1: Kalman Filter and Retrodiction

3.1 Maximum Likelihood Estimate

The Kalman filter has differing versions like the extended and unscented Kalman filters, and so does retrodiction. This presents a slight dilemma as to which Smoothing filter should be chosen for the task ahead. Since the point is to break new ground, it really doesn't matter which algorithm is selected, since the work done here will be easily ported to other areas. As a result, it seemed sensible to start with one of the more popular (albeit older) versions of retrodiction. With that said, this thesis will concentrate its efforts on deriving a parallel algorithm for the retrodiction of the Kalman filter as defined by H. E. Rauch, F. Tung and C. T. Striebel [34]. Their paper was published shortly after Kalman's and is considered by far to be the originator of

the Smoothing filter.

The problem of Smoothing involves using state and covariance estimates that have already been obtained by the Kalman filter. Since one must compute the covariance at a given time in order to compute the estimate, it is assumed that the covariance matrix computed at each and every time step of the Kalman filter has been retained in memory. With both the estimated covariance matrices and the state estimation vectors, one can more easily carry out a parallel implementation of the Smoothing problem. In their work entitled: Maximum Likelihood Estimates of Linear Dynamic Systems [34], the authors were able to develop the discrete time solution to the problem of Smoothing, which will be reviewed here.

The goal is to find the optimal estimate $\hat{x}_{k|N}$ given the data set z_0,z_1,...,$z_N = Z_k$, where N is the size of the entire data set under consideration. If $k = N$ it is a filtering problem, if $k > N$ it is a prediction problem and if $k < N$ it is a retrodicton problem. The case where $k > N$ was discussed in depth throughout the previous section, and here the objective is to solve for the case when $k < N$.

According to the maximum likelihood estimate

$$L(x_k, x_{k+1}, Z_N) = \log p(x_k, x_{k+1}|Z_k) \tag{3.1.1}$$

If the estimates $\hat{x}_{k|k}$ have already been computed by the Kalman filter, then (3.1.1) can be expressed as

$$\begin{aligned} \max_{x_k, x_{k+1}} L(x_k, x_{k+1}, Z_N) = \max_{x_k, x_{k+1}} \{ & -\|x_{k+1} - F_k x_k\|^2 Q_k^{-1} \\ & -\|x_k - \hat{x}_{k|k}\|^2 P_{k|k}^{-1}\} \end{aligned} \tag{3.1.2}$$

where the terms from (3.1.1) that do not contain x_k have been omitted. Replacing the first term in (3.1.2) with the desired estimate gives

$$J = -\|\hat{x}_{k|N} - F_k x_k\|^2 Q_k^{-1} + \|x_k - \hat{x}_{k|k}\|^2 P_{k|k}^{-1} \tag{3.1.3}$$

Taking the derivative with respect to x_k, setting it equal to zero and solving for $\hat{x}_{k|N}$ produces

$$\hat{x}_{k|N} = \hat{x}_{k|k} + P_{k|k} F_k^T P_{k|k}^{-1} [\hat{x}_{k+1|N} - F_k \hat{x}_{k|k}] \tag{3.1.4}$$

The covariance can then be calculated and shown to be

$$P_{k|N} = P_{k|k} + C_k [P_{k+1|N} - P_{k+1|k}] C_k^T \tag{3.1.5}$$

where

$$\begin{aligned} C_k &= P_{k|k} F_k^T [F_k P_{k|k} F_k^T + Q_k]^{-1} \\ &= P_{k|k} F_k^T P_{k|k}^{-1} \end{aligned} \tag{3.1.6}$$

Equations (3.1.4) and (3.1.5) are the solution to the retrodiction problem, and it is these two equations that form the basis of the parallization algorithm in the pages to follow. Although the proof outlined from (3.1.1) through to (3.1.5) was not thoroughly comprehensive (the reader is referred to [34] for a detailed overview), it was enough to emphasize how the prediction estimates $\hat{x}_{k|k}$ affect the smoothed state.

3.1.1 One Iteration II

As was the case in section 2.1, this section will step through the necessary matrix operations in order to appreciate what is involved in computing the retrodiction of the Kalman filter. It makes sense to start the calculations by seeking a solution to C_k, since this matrix is needed in order to update both the state and covariance estimates. However, in order to solve for C_k, both $P_{k|k}$ and $P_{k+1|k}$ have to be calculated. It is possible to store $P_{k|k}$ in memory during the Kalman filter computations, or to compute it using $P_{k+1|k+1}$, the latter will be illustrated below as a matter of conciseness.

$$P_{k+1|k} = (P^{-1}_{k+1|k+1} - H^T_{k+1} R^{-1}_k H_k)^{-1} \tag{3.1.1.1}$$

If it seems distasteful to have to solve an inverse for 3 distinct matrices, then the above solution can be modified to

$$\begin{aligned} P_{k+1|k} = P_{k+1|k+1} - P_{k+1|k+1} H^T_{k+1} (H_{k+1} P_{k+1|k+1} H^T_{k+1} \\ - R_k)^{-1} H_k P_{k+1|k+1} \end{aligned} \tag{3.1.1.2}$$

In either case the sought after solution is

$$P_{k|k} = F_k (P_{k+1|k} - Q_k) F_k \tag{3.1.1.3}$$

Deriving C_k is then a trivial matter of substituting the equation above into (3.1.6).

There is something to note concerning the covariance matrix $P_{k+1|k}$, and the subscripts accompanying it. Since the retrodiction of the Kalman filter first requires the computation of the state and covariance estimates of the Kalman filter itself, and in calculating the estimation covariance it would be necessary to compute $P_{k+1|k}$, the

reader might be inclined to consider the possibility of retaining the matrix in memory for later use by the Smoothing algorithm. This, however, is not possible since the matrix $P_{k+1|k}$ in the Smoothing problem is not the same matrix defined in (2.1.8).

An important thing to keep in mind is that the algorithm begins with the last estimate calculated by the Kalman filter, which means that if the data set is of length N, then $k+1 = N, N-1, ..., 1$. Therefore, the Smoothing problem begins with the last estimation covariance computed

$$P_{k+1|N} = P_{k+1|k+1} \tag{3.1.1.4}$$

where $P_{k+1|k+1}$ is the estimated covariance at time $k+1 = N$. All of the necessary components required to compute the updated estimation covariance $P_{k|N}$ in (3.1.5) are now accounted for. Similarly, all of the values needed for solving (3.1.4) are also present, except for $\hat{x}_{k+1|N}$, which is nothing more than

$$\hat{x}_{k+1|N} = \hat{x}_{k+1|k+1} \tag{3.1.1.5}$$

Once $\hat{x}_{k|N}$ and $P_{k|N}$ are computed using (3.1.4) and (3.1.5), respectively, then they can be substituted back into the aforementioned formulas by setting them equal to

$$\hat{x}_{k+1|N} = \hat{x}_{k|N} \tag{3.1.1.6}$$

and

$$P_{k+1|N} = P_{k|N} \tag{3.1.1.7}$$

A cautionary word to the reader; equations (3.1.1.5) and (3.1.1.4) are only used

to initialize $\hat{x}_{k+1|N}$ and $P_{k+1|N}$, while (3.1.1.6) and (3.1.1.7) are used to update the variables every iteration thereafter.

3.1.2 Computational Cost II

The number of matrix/vector operations needed in order to assess the runtime complexity of the Smoothing problem is best illustrated using Table 3.1. Once again, H_k^T and $P_{k+1|k+1}H_k^T$ were counted twice, so the actual number of matrix/vector computations is probably 25 as opposed to 27. Furthermore, in the table below the value of $P_{k+1|k}$ was computed according to (3.1.1.2), which contains 9 matrix operations. If (3.1.1.1) were used instead, only 7 matrix computations would be required bringing the total count to 23, bearing in mind that 3 of those computations are inverse operations. Lastly, there are 3 matrix computations required to solve for $P_{k|k}$, however, this can be ignored if the estimation covariance calculated during the Kalman filter was retained in memory, which means there are 20 matrix/vector computations. No matter what though, the retrodiction of the Kalman filter needs a bit more work than the prediction, which only justifies the need for a parallel algorithm even more.

No.	Operation	Variables	Runtime
1	Multiplication	$H_{k+1}P_{k+1\|k+1}$	$O(n_s^2 n_d^3)$
2	Transpose	H_{k+1}^T	$O(n_s n_d^2)$
3	Multiplication	$H_{k+1}P_{k+1\|k+1}H_{k+1}^T$	$O(n_s^2 n_d^3)$
4	Subtraction	$H_{k+1}P_{k+1\|k_1}H_{k+1}^T - R_{k+1}$	$O(n_d^2)$
5	Inverse	$(H_{k+1}P_{k+1\|k+1}H_{k+1}^T - R_{k+1})^{-1}$	$O(n_d^3)$
6	Multiplication	$H_{k+1}P_{k+1\|k+1}$	$O(n_s^2 n_d^3)$
7	Multiplication	$(H_{k+1}P_{k+1\|k+1}H_{k+1}^T - R_{k+1})^{-1}$	

		$H_{k+1}P_{k+1\|k+1}$	$O(n_s n_d^3)$
8	Transpose	H_{k+1}^T	$O(n_s n_d^2)$
9	Multiplication	$P_{k+1\|k+1}H_{k+1}^T$	$O(n_s^2 n_d^3)$
10	Multiplication	$P_{k+1\|k+1}H_{k+1}^T$	
		$(H_{k+1}P_{k+1\|k+1}H_{k+1}^T - R_{k+1})^{-1}$	
		$H_{k+1}P_{k+1\|k+1}$	$O(n_s n_d^3)$
11	Subtraction	$P_{k+1\|k+1} - P_{k+1\|k+1}H_{k+1}^T$	
		$(H_{k+1}P_{k+1\|k+1}H_{k+1}^T - R_{k+1})^{-1}$	
		$H_{k+1}P_{k+1\|k+1}$	$O((n_s n_d)^2)$
12	Subtraction	$P_{k+1\|k} - Q_k$	$O((n_s n_d)^2)$
13	Multiplication	$F_k(P_{k+1\|k} - Q_k)$	$O((n_s n_d)^3)$
14	Multiplication	$F_k(P_{k+1\|k} - Q_k)F_k$	$O((n_s n_d)^3)$
15	Transpose	F_k^T	$O((n_s n_d)^2)$
16	Multiplication	$P_{k\|k}F_k^T$	$O((n_s n_d)^3)$
17	Inverse	$P_{k\|k}^{-1}$	$O((n_s n_d)^3)$
18	Multiplication	$P_{k\|k}F_k^T P_{k\|k}^{-1}$	$O((n_s n_d)^3)$
19	Subtraction	$P_{k+1\|N} - P_{k+1\|k}$	$O((n_s n_d)^2)$
20	Transpose	C_k^T	$O((n_s n_d)^2)$
21	Multiplication	$(P_{k+1\|N} - P_{k+1\|k})C_k^T$	$O((n_s n_d)^3)$
22	Multiplication	$C_k(P_{k+1\|N} - P_{k+1\|k})C_k^T$	$O((n_s n_d)^3)$
23	Addition	$P_{k\|k} + C_k(P_{k+1\|N} - P_{k+1\|k})C_k^T$	$O((n_s n_d)^2)$
24	Multiplication	$F_k\hat{x}_{k\|k}$	$O((n_s n_d)^2)$
25	Subtraction	$\hat{x}_{k+1\|N} - F_k\hat{x}_{k\|k}$	$O(n_s n_d)$
26	Multiplication	$C_k(\hat{x}_{k+1\|N} - F_k\hat{x}_{k\|k})$	$O((n_s n_d)^2)$

27	Addition	$\hat{x}_{k\|k} + C_k(\hat{x}_{k+1\|N} - F_k\hat{x}_{k\|k})$	$O(n_s n_d)$

Table 3.1: Operational Complexity of the Smoothing Problem

Looking at the table above and it is clear that the largest runtime is $O((n_s n_d)^3) \approx O(n^3)$. Recall from the last chapter that the cubic runtime of the Kalman filter was the primary motivation for using a GPU, and that motivation applies here, as well.

3.2 Derivation of a Parallel Algorithm

There is no doubt that the procedure in chapter 2 offered an increase in performance when compared to a serial program. Yet, it merely showed how to configure basic matrix multiplication into block form and implement that onto a computer architecture better suited for it. The real prize would be to not only exploit the properties of a GPU by more efficiently organizing matrix operations, but to convert the algorithm into a genuine parallel algorithm. The first question that should be posed is whether or not this has been attempted before? The answer is yes [25] [24], but only in a limited capacity. The authors of the publication entitled; Parallel Smoothing, have opted to divide the data set into subintervals, and perform 3 distinct steps. The first step involves performing computations on each subinterval simultaneously, using independent processors. The second step demands that the smoothed estimates at the boundaries of each interval are combined in order to smooth them out. The final step requires yet, another retrodiction performed on the all the data points in order to obtain a global solution. The algorithm proposed here is entirely different, since the recursive properties have been replaced with the prefix sums algorithm that has a

logarithmic runtime, and the computations are performed using the same block operations detailed in chapter 2. This suggests that the retrodiction should out perform the prediction since it incorporates the strategy from section 2.2.3 while simultaneously adopting the prefix sums algorithm.

Recall that in section 2.2, the matter of applying the prefix sums algorithm outlined in [10] was said to be impractical, but such is not the case with the retrodiction of the Kalman filter. The difference being that the Kalman filter would require all the previous estimates in order to predict the successive one and as was mentioned earlier, computing 20 estimates just to predict the 21^{st} is wasteful, while retrodiction relies on already knowing all the past predictions. This is the key feature that distinguishes retrodiction from prediction when vying for potential candidates that can be paralellized.

3.2.1 Prefix Sums Operation

Deriving a parallel algorithm from a recursive one is not such a difficult task if one has the right tools. The problem is that parallel processing units are relatively new, and so the documentation that algorithm designers need to aid them in exploiting this new technology is equally modern and therefore, not as plentiful. Yet, the last several years have seen the arrival of core building blocks that designers will need if they are to thrive in the field of GPU software development. The most popular of these being the all-prefix-sums algorithm. Throughout this thesis there has been mention of the prefix sums algorithm, and how it should be considered when attempting to parallelize a serial algorithm. Its importance has been well embellished in the previous pages, and now, finally, its powers will be revealed.

Referring to [10], the all-prefix-sums operation takes a binary associative operator $\oplus$ and an ordered set of n elements $[a_0, a_1, ..., a_{n-1}]$ and returns the ordered set $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 ... \oplus a_{n-1})]$. For example, if the binary operation is addition, then the input

$$\begin{bmatrix} 3 & 5 & 1 & 1 & 0 & 2 & 4 & 1 \end{bmatrix}$$

will produce the output

$$\begin{bmatrix} 3 & 8 & 9 & 10 & 10 & 12 & 16 & 17 \end{bmatrix}$$

And if the binary operation is multiplication, then the same input from above would produce the following output

$$\begin{bmatrix} 3 & 15 & 15 & 15 & 0 & 0 & 0 & 0 \end{bmatrix}$$

If the all-prefix-sums operation is performed on a vector, then it is known as the scan operation. Performing the scan on a vector using a CPU is simple enough, all that is required is to loop through the array adding all the elements. It shouldn't be too hard to convince the reader that the runtime complexity of such an algorithm is $O(n)$. The important thing to note is that the code for this algorithm would be executed by a single thread, so how does one exploit the benefit of having multiple threads? A simple, yet elegant way of doing this is outlined below.

The computation begins by adding every value located at an even index (assuming the indices begin at 0) to the adjacent value located at an odd index, wherein the odd indices must be greater than the even indices. This is displayed by the arrows of Figure 3.2, which are denoted by d1. Notice that d1 does not represent the row, but

the operations between rows, i.e. the arrows. The sum of these values are stored in the odd indices. The next set of operations is d2, where the values from the previous iteration are summed with their nearest neighbour, which must be located at one of the odd indices from the previous iteration. The only constraints are that the value located at the smallest index is added to its neighbour and no value can be added to two different neighbours. This continues upward until there is only one value remaining.

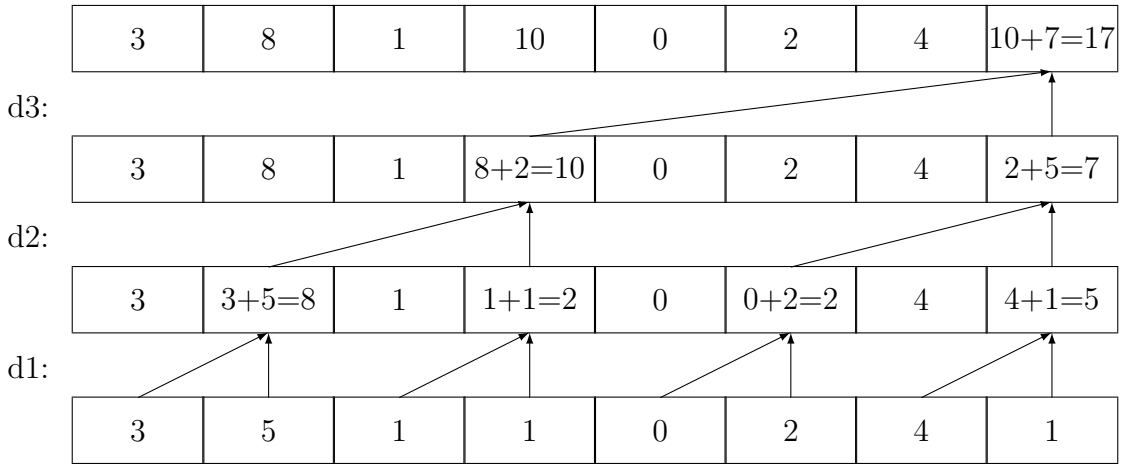

Figure 3.2: Up-Sweep Scan

The important thing to note when referring to Figure 3.2 is that d1 is performed in one FLOP on a GPU, wherein a CPU would require 4 FLOPS. In fact, d1, d2 and d3 each require 1 FLOP, where in this particular example the CPU would need to perform 7 FLOPS. This may not seem like an impressive improvement, but what if the length of the array was not 8, but 1,000? Since the up-sweep reduces the number of necessary array elements by half every iteration, then the number of FLOPS required to execute the up-sweep by a GPU is $\log_2 n$. Therefore, running the up-sweep on an array of 1,000 elements would require 10 FLOPS from a GPU, and 999 from a CPU.

One can clearly see how this could quickly be advantageous for large sized arrays. Yet this is only half of the algorithm, the down-sweep needs to be performed as well.

The down-sweep begins by taking the resultant array of the up-sweep and setting the last index equal to 0, which is indicated by d0 in Figure 3.3. Then the rest of the algorithm is just the up-sweep in reverse. In the final computation of the up-sweep the value at index 7 was added to that at index 3, and in the down-sweep this is the first operation. There is one additional operation that requires careful consideration and that is before the new value is stored in index 7, the old value must be swapped to index 3. So at every iteration addition is performed and swaps are made. Upon close inspection the reader can verify that all of the indices involved in the up-sweep are also engaged in the down-sweep. The final result is identical to the expected answer except that the every value has been shifted by one index to the right. This is not a problem since they can be shifted back and the last value calculated in the up-sweep can be temporarily stored in a variable and set back into the last index once the array has been shifted.

The runtime of the up-sweep was determined to be $O(\log_2 n)$. Since the down-sweep contains the same number of operations except for the additional swaps, it should have a complexity of $O(\log_2 n)$, which gives a total runtime of $O(\log_2 n)$.

The trivial example outlined in this section may not seem very insightful or practical for problems that contain complex calculations like the Smoothing problem, after all, how does summing a bunch of numbers help when performing 27 distinct matrix/vector computations? The answer is that if an algorithm, not matter how intricate, can be proven to meet certain requisites, then the scan algorithm can be implemented. The proof of this is undoubtedly the most novel aspect and the greatest

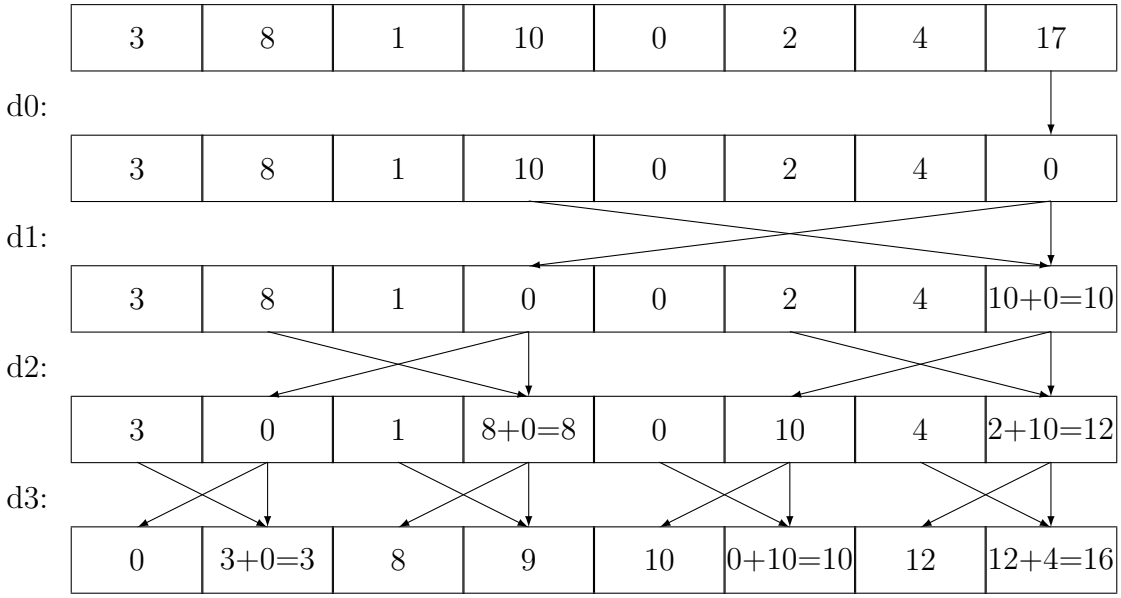

Figure 3.3: Down-Sweep Scan

contribution of this thesis.

3.2.2 Parallelizing Complex Functions

In order to derive a parallel algorithm from a recursive one, the first step is to prove that it meets the criteria outlined in the following equations

$$x_i = \begin{cases} b_0 & i = 0 \\ (x_{i-1} \otimes a_i) \oplus b_i & 0 < i < N \end{cases} \tag{3.2.2.1}$$

where a_i and b_i are arbitrary constants and $\otimes$ and $\oplus$ are binary operators [16]. The key to converting a recursive algorithm into a parallel one is by way of the constraints

defined for the binary operators, namely

$$\oplus \text{ is associative} \tag{3.2.2.2a}$$

$$\otimes \text{ is semi-associative} \tag{3.2.2.2b}$$

$$\otimes \text{ is distributive} \tag{3.2.2.2c}$$

As long as these three properties are satisfied, then (3.2.2.1) can be converted into a parallel algorithm. However, it is the intent of this thesis to go one step further and implement the parallel algorithm on an actual GPU, which requires the use of the scan operation, defined by the following equations

$$x_i = \begin{cases} b_0 & i = 0 \\ x_{i-1} \otimes a_i & 0 < i < N \end{cases} \tag{3.2.2.3}$$

and an ordered set that obeys the following recursive formula

$$y_i = \begin{cases} a_0 & i = 0 \\ y_{i-1} \odot a_i & 0 < i < N \end{cases} \tag{3.2.2.4}$$

The operator $\odot$ is referred to as the companion operator of the binary operator $\otimes$, and is defined as a binary associative operator that satisfies the following

$$(a_i \otimes b_i) \otimes c_i = a_i \otimes (b_i \odot c_i) \tag{3.2.2.5}$$

Furthermore, if $\otimes$ is fully associative then it is equal to $\odot$. The manner in which (3.2.2.1) can be reduced to (3.2.2.3) is by way of, yet another binary operator defined

as

$$c_i \bullet c_j = [c_{i,a} \odot c_{j,a}, (c_{i,b} \otimes c_{j,a}) \oplus c_{j,b}] \tag{3.2.2.6}$$

wherein $c_{i,a}$ and $c_{i,b}$ are elements belonging to the set c_i. So if the criteria outlined in equations (3.2.2.1) to (3.2.2.6) are met, then a parallel algorithm can be derived from a recursive one and implemented using the scan algorithm outlined in [10].

3.2.3 The Smoothed States

Armed with the formulas outlined in section 3.2.2, the derivation of the parallel algorithm begins by expanding an rearranging (3.1.4)

$$\hat{x}_{k|N} = \hat{x}_{k|k} - C_k F_k \hat{x}_{k|k} + C_k \hat{x}_{k+1|N} \tag{3.2.3.1}$$

Now let

$$b_k = \hat{x}_{k|k} - C_k F_k \hat{x}_{k|k} \tag{3.2.3.2}$$

$$a_k x_{k+1} = C_k \hat{x}_{k+1|N} \tag{3.2.3.3}$$

and

$$x_k = \hat{x}_{k|N} \tag{3.2.3.4}$$

The result is an equation much like that found in (3.2.2.1)

$$x_k = a_k x_{k+1} + b_k \tag{3.2.3.5}$$

What this means is that (3.2.3.1) is in the appropriate form needed to parallelize it. Now that the equation is in the proper form, the binary operators can clearly be

classified as vector addition

$$a_k x_{k+1} + b_k = (a_k x_{k+1}) \oplus b_k \tag{3.2.3.6}$$

and matrix-vector multiplication

$$a_k x_{k+1} = x_{k+1} \otimes a_k \tag{3.2.3.7}$$

According to the previous section, the three properties, defined in (3.2.2.2a), (3.2.2.2b) and (3.2.2.2c) have to be satisfied. The first property (3.2.2.2a), is the easiest to prove and quite simply equates to the fact that matrix/vector addition is fully associative.

$$(a \oplus b) \oplus c = a \oplus (b \oplus c) \tag{3.2.3.8}$$

The second property (3.2.2.2b), is proven by first noticing that the binary operation in (3.2.3.7) is the equivalent of the transpose of matrix/vector products

$$\begin{aligned} (a \otimes b) \otimes c &= c(ba) \\ &= (cb)a \\ &= a \otimes (cb) \end{aligned} \tag{3.2.3.9}$$

It is easy enough to assert that the transpose of matrix/vector products is the product of transposed matrix/vectors, provided the dimensions agree. Therefore, if u and v are appropriately sized matrices/vectors then the following property applies

$$(uv)^T = v^T u^T \tag{3.2.3.10}$$

Extending this idea, it can be verified that the transpose of matrix/vector products is associative by first acknowledging that matrix/vector multiplication is associative. Therefore, given 3 matrices/vectors u, v and w, with the appropriate dimensions, it can easily be verified that

$$\begin{aligned}((uv)w)^T &= w^T(v^T u^T) \\ &= (w^T v^T)u^T\end{aligned} \tag{3.2.3.11}$$

This proves that not only is the transpose of matrix/vector products associative, it is fully associative. So if $\otimes$ is fully associative then $\odot$ is as well and (3.2.3.9) is equal to

$$\begin{aligned}a \otimes (cb) &= a \otimes (b \odot c) \\ &= a \otimes (b \otimes c)\end{aligned} \tag{3.2.3.12}$$

This has the added benefit of proving that (3.2.2.4) is associative as well. The final requirement is to satisfy (3.2.2.2c), which can easily be done through the following steps

$$\begin{aligned}a \otimes (b \oplus c) &= (b \oplus c)a \\ &= (ba \oplus ca) \\ &= ((a \otimes b) \oplus (a \otimes c))\end{aligned} \tag{3.2.3.13}$$

Hence, the $\otimes$ operator is distributive, and therefore the recursive equation in (3.2.3.5) satisfies all three requirements, and as such can be implemented in a parallel algorithm. This does not, however, mean that it can be used in the scan algorithm described in [10], which is the final goal.

Two more steps need to be taken, first (3.2.2.1) must be reduced to (3.2.2.3) and second an ordered set must satisfy (3.2.2.4). The first step can be accomplished by fulfilling the necessity of showing that the binary operator $\bullet$ is associative given the following set

$$c_i = [a_i, b_i] \tag{3.2.3.14}$$

Therefore, the following must be proven

$$(c_i \bullet c_j) \bullet c_k = c_i \bullet (c_j \bullet c_k) \tag{3.2.3.15}$$

Just to be clear, the values c_i, c_j and c_k represent the arbitrary constants defined in (3.2.2.1) for differing sampling times i, j and k. Substituting the definition defined in (3.2.2.6) into the above equation results in

$$\begin{aligned}
(c_i \bullet c_j) \bullet c_k &= [c_{i,a} \odot c_{j,a}, (c_{i,b} \otimes c_{j,a}) \oplus c_{j,b}] \bullet c_k \\
&= [(c_{i,a} \odot c_{j,a}) \odot c_{k,a}, (((c_{i,b} \otimes c_{j,a}) \oplus c_{j,b}) \otimes c_{k,a}) \oplus c_{k,b}] \\
&= [(c_{j,a}c_{i,a}) \odot c_{k,a}, ((c_{j,a}c_{i,b}) + c_{j,b}) \otimes c_{k,a} + c_{k,b}] \\
&= [(c_{k,a}c_{j,a})c_{i,a}, c_{k,a}(c_{j,a}c_{i,b}) + c_{k,a}c_{j,b} + c_{k,b}] \\
&= [c_{i,a} \odot (c_{j,a} \odot c_{k,a}), (c_{k,a}c_{j,a})c_{i,b} + c_{k,a}c_{j,b} + c_{k,b}] \\
&= [c_{i,a} \odot (c_{j,a} \odot c_{k,a}), c_{i,b} \otimes (c_{j,a} \otimes c_{k,a}) \oplus c_{j,b} \otimes c_{k,a} \oplus c_{k,b}]
\end{aligned} \tag{3.2.3.16}$$

It has already been proven in (3.2.3.13) that $\otimes$ is distributive, therefore

$$\begin{aligned}
&[c_{i,a} \odot (c_{j,a} \odot c_{k,a}),\, c_{i,b} \otimes (c_{j,a} \otimes c_{k,a}) \oplus c_{j,b} \otimes c_{k,a} \oplus c_{k,b}] \\
&\qquad = c_i \bullet [c_j \odot c_k,\, (c_{j,b} \otimes c_{k,a}) \oplus c_{k,b}] \\
&\qquad = c_i \bullet (c_j \bullet c_k)
\end{aligned} \tag{3.2.3.17}$$

Therefore the $\bullet$ operator is associative. The second step to finalizing the parallelization of the estimates starts by establishing an ordered set

$$s_k = [y_k, x_k] \tag{3.2.3.18}$$

The variable y_k was already defined in (3.2.2.4), and so all that remains is to prove that this recursive property holds. The initialization of the ordered set is rather straightforward.

$$\begin{aligned}
s_N &= [y_N, x_N] \\
&= [a_N, b_N] \\
&= c_N
\end{aligned} \tag{3.2.3.19}$$

Proof that the binary operators outlined in equations (3.2.3.6) and (3.2.3.7) will work

on the ordered set begins by mixing the ordered set with the constants from (3.2.2.1).

$$\begin{aligned} s_k &= [y_k, x_k] \\ &= [y_k, a_k x_{k+1} + b_k] \\ &= [y_{k+1} \odot a_k, (x_{k+1} \otimes a_k) \oplus b_k] \\ &= [y_{k+1} \odot c_{k,a}, (x_{k+1} \otimes c_{k,a}) \oplus c_{k,b}] \\ &= [y_{k+1}, x_{k+1}] \bullet c_k \\ &= s_{k+1} \bullet c_k \end{aligned} \tag{3.2.3.20}$$

Recall that the $\bullet$ operator was established as being associative in (3.2.3.17), and as a result (3.2.2.1) can be reduced to (3.2.2.3). This concludes the proof for the estimates within the Smoothing problem. It has been shown that not only can a parallel algorithm be derived, but it can be implemented on a parallel processor using the scan algorithm described in [10].

3.2.4 The Smoothed Covariance

The next portion of the proof consists of applying the same series of steps implemented on the estimates to the covariance matrix in (3.1.5). As with the estimates, the equation in question is expanded and rearranged

$$P_{k|N} = P_{k|k} - C_k P_{k+1|k} C_k^T + C_k P_{k+1|N} C_k^T \tag{3.2.4.1}$$

The first thing to note is how the covariance matrix $P_{k+1|N}$ is wedged between the two matrices C_k and C_k^T. This presents a problem when attempting to get the equation into the form dictated by (3.2.2.1). The solution to this obstacle is an elegant, yet

simple one that consists of a two step process, which begins with the *vec* operator [27] that enables one to turn a matrix into a stacked vector of the columns of that matrix. Refer to Appendix B for further details.

Applying the *vec* operator to both sides of (3.2.4.1), results in

$$vec(P_{k|N}) = vec(P_{k|k} - C_k P_{k+1|k} C_k^T) + vec(C_k P_{k+1|N} C_k^T) \tag{3.2.4.2}$$

Assuming all of the matrices in the above equation are $n \times n$, then

$$vec(P_{k|N}) \quad \text{has dimensions } n^2 \times 1 \tag{3.2.4.3a}$$

$$vec(P_{k|k} - C_k P_{k+1|k} C_k^T) \quad \text{has dimensions } n^2 \times 1 \tag{3.2.4.3b}$$

$$vec(C_k P_{k+1|N} C_k^T) \quad \text{has dimensions } n^2 \times 1 \tag{3.2.4.3c}$$

The last expression (3.2.4.3c) is problematic, and thus requires the use of the second step, which is the Kronecker product defined in Appendix B, and stated here for convenience.

$$vec(AXB) = (B^T \bigotimes A) vec(X) \tag{3.2.4.4}$$

It is important to note that the $\bigotimes$ operator above is the Kronecker product and not the same binary operator $\otimes$ that was applied in the previous sections. This is an unfortunate consequence of notation, and as a result this paper will adopt the boldface notation $\bigotimes$ to mean the Kronecker product, so as to differentiate between the two.

The expression from (3.2.4.3c) can now be configured using (3.2.4.4) in the following manner

$$vec(C_k P_{k+1|N} C_k^T) = (C_k \bigotimes C_k) vec(P_{k+1|N}) \tag{3.2.4.5}$$

where

$vec(P_{k+1\|N})$	has dimensions $n^2 \times 1$
$(C_k \bigotimes C_k)$	has dimensions $n^2 \times n^2$

The same strategy that was applied to the estimates in the last section is adopted here, wherein each respective component, namely (3.2.4.3a), (3.2.4.3b) and (3.2.4.5) is substituted into (3.2.3.5)

$$x_{k,P} = vec(P_{k|N}) \tag{3.2.4.7}$$

$$b_{k,P} = vec(P_{k|k} - C_k P_{k+1|k} C_k^T) \tag{3.2.4.8}$$

$$a_{k,P} x_{k+1,P} = (C_k \bigotimes C_k) vec(P_{k+1|N}) \tag{3.2.4.9}$$

where P in the subscripts above is used to distinguish them from the equations used to represent the state estimates in (3.2.3.2) to (3.2.3.4).

Now they are in a form that is identical to (3.2.3.5), in fact, even the binary operators are the same. Therefore, all 3 of the properties in (3.2.2.2) hold true, as well as the subsequent proofs that showed (3.2.2.1) can be reduced to (3.2.2.3) and the ordered set in (3.2.2.4) does exist. Therefore, the parallelization of the estimation covariance matrices are both possible and capable of being carried out via the scan operation explained in section 3.2.2.

The sole difference between the equations outlined in (3.2.3.2) to (3.2.3.4) and

those from (3.2.4.7) to (3.2.4.9) are that the dimensions differ by an entire magnitude. This, however does not come as a surprise nor a problem. The estimated state vector is always an entire magnitude smaller than the estimated covariance matrix. Whether one adds two vectors of size $n^2 \times 1$ or two matrices with dimensions $n \times n$, makes no difference to the computational intensity. Even when solving for (3.2.4.9), the runtime complexity is not adversely affected. This issue will be addressed in more detail in the following section.

3.2.5 Parallel Computational Cost

In section 2.1.1 it was shown that the most significant runtime of the Kalman filter was $O(n^3)$, which when compared to the complexity for the parallel algorithm of $O(n)$ determined in section 2.2.3, justified parallelizing the computations of the Kalman filter. The situation may prove to be quite similar when considering the Smoothing algorithm. All of the matrix computations outlined in Table 3.1 are within the same dimensional range of those shown in Table 2.1,and the algorithm derived in section 3.2.4 only suggests a slightly different outcome from that derived in section 2.2.3. Specifically, that defined in (3.2.4.5), which shows that the Kronecker product transforms $C_k \bigotimes C_k$ into an $n^2 \times n^2$ matrix, while the *vec* operator turns $P_{k+1|N}$ into an $n^2 \times 1$ vector. This matrix and vector will have to be multiplied by one another, but fortunately that only equates to a matrix-vector product whose runtime can be greatly diminished with a GPU. For example, using a GPU to perform matrix-vector multiplication of an $n \times n$ matrix and an $n \times 1$ vector will result in a constant runtime of $O(1)$, provided there are $n \times n$ threads operating within the block. Using the same number of threads for an $n^2 \times n^2$ matrix times an $n^2 \times 1$ vector will cost $O(n)$.

The added computation of having to multiply the resulting Kronecker product with $vec(P_{k+1|N})$ will increase the execution time but not the code complexity, which will ultimately remain linear. This is but one consideration, the other being the actual Kronecker product, which will consist of multiplying a matrix times a scalar. If the matrix has dimensions 4×4, then there will have to be 16 matrix-scalar operations, wherein each operation has a complexity of $O(1)$. This may seem like a trivial matter, but as will be seen in the next section, there are only 64 blocks, and so only 4 matrices at a time can be computed, which will cause serialization. That will amount to 16 FLOPS required to compute the Kronecker product for 64 matrices.

At this point it might be worth mentioning that it is possible to dismiss the parallelization of the estimation covariance. The results of $P_{k|N}$ can be acquired using block matrix operations similar to those shown in section 2.2.3, in which case the *vec* operator will not be utilized. The price of doing this will restrict the designer from using the scan operation on the estimated covariance, which will in turn serialize the estimated covariance calculations, but not those of the state estimation.

The good news is that one has options. One can ignore the use of the scan operation, in which case sections 3.2.3 and 3.2.4 no longer apply and block operations may be used to solve the Smoothing problem serially. Or one can incorporate the algorithm in section 3.2.3 while ignoring that from section 3.2.4, or both sections 3.2.3 and 3.2.4 can be integrated onto a GPU. One final option is to ignore the calculation of the estimated covariance all together. The solution to the Smoothing problem is primarily concerned with the state estimates, and the covariance estimates may prove useless. The reason this is even possible is obvious when one looks at 3.2.3.1, wherein the retrodiction estimate $\hat{x}_{k|N}$ is in no way dependant on the updated covariance

$P_{k|N}$. With that said, the sole question that remains is which of the 4 aforementioned options produces the best results?

3.2.6 The Source Code II

The results produced in section 2.2.6 were based upon the example outlined in section 2.2.3, wherein each block contained 16 matrices, the largest of which was 6×6. The size of each matrix was a result of there being 3 state space variables (position, velocity and acceleration) and 2 dimensions (x and y). The following simulation will be slightly different. Instead, there will be 2 state space variables, namely position and velocity, and 2 dimensions, x and y. Furthermore, the implementation of the algorithm will not be executed as outlined in section 2.2.3, wherein each block consisted of 24×24 threads subdivided into 16 matrices, instead each block will consist 4×4 threads, since the largest matrix is of size 4×4, and there will be a total of 64 blocks. The reasoning behind this change is simplicity. Subdividing a block into smaller matrices is quite difficult and time consuming, whereas allocating a matrix for each block greatly simplifies the task of coding. It comes down to performance versus complexity; is it really worth while to cram 16 matrices into a block in order to increase thread performance? This is an important question that every programmer must face. Surely, utilizing more threads will increase performance, but by how much? If it takes an extra month to code the algorithm while seeing an increase in functionality by a fractional amount, then it might not be worth one's while. Therefore, this section resolves two questions at once, the first is how much of a speed up does the algorithm presented in section 3.2.3 offer, and the second being how much does using a reduced thread count impeded performance?

The code was structured much like what was seen in section 2.2.4. The first consideration is whether or not shared memory will enhance performance. In this particular case there are a substantial amount of matrix computations that will benefit from the use of shared memory and therefore, allocating the appropriate amount of shared memory is the first step. The programmer must ensure that every matrix/vector is allocated enough shared memory to store 64 matrices/vectors appropriately sized for 2 state space variables and 2 dimensions. For example, the state estimation vector $\hat{x}_{k|k}$ should be allocated 256 array elements, while the state transition matrix should be given 1,024 array elements, both of type `float`.

There is one major difference between the code that was outlined in section 2.2.4 and that which is defined here, that being the fact that threads between differing blocks cannot interact. Threads from block 3 cannot be added, subtracted, multiplied, divided or in any way arithmetically combined with those from any other block. This causes a problem when performing both the up-sweep and the down-sweep of the scan since b_k (refer to equation 3.2.3.2) will reside in block k and have to be added to b_{k+1} located on block $k+1$. The workaround for this is to compute b_k using shared memory and once that is complete the results can be transferred into global memory where the values residing on different blocks can be arithmetically interchanged. This, however, adds additional transfers from shared to global memory and draws into question the efficiency of using shared memory. The only way to know if shared memory is better equipped for the algorithm in question is to write a second piece of code that uses global memory only, and compare that to the performance of the aforementioned code. This is well out of the scope of this thesis, since the goal is to derive a parallel algorithm using the prefix sums and not to explore every avenue of

possible optimization. Therefore, the option of writing code that solely utilizes global memory will have to be tucked away for future considerations.

As was mentioned above, the matrix computations will be divided into two distinct categories; those that are executed in shared memory and those using global memory. The first category will be executed exactly as was seen in Listing 2.1, wherein two nested `for` loops will be utilised to multiply the columns and rows of the matrices in question, while only one `for` loop is required to multiply a matrix by a vector. The idea is to perform matrix/vector operations until every value of C_k and b_k in shared memory has been computed, and afterwards the results can be transferred into global memory and the scan algorithm can begin.

As a matter of conciseness, it seems prudent to elaborate on the exact methodology that will be utilised in performing the up and down-sweep. One of the key issues is to solve for b_k and C_k serially, prior to the scan algorithm being implemented. That means b_k and C_k must be computed for all values of $k = 1, ..., N$ and once they are calculated then the up-sweep can begin

It is at this point that the reader can appreciate all of the effort that went into exploring matrix block operations in the previous chapter. These block operations can now be easily assimilated by the present chapter when solving for b_k and C_k, which begins by looking at how b_{k+1} is updated.

$$b^*_{k+1} = b_{k+1} + C_{k+1}b_k \tag{3.2.6.1}$$

and C_{k+1} is updated as follows

$$C^*_{k+1} = C_{k+1}C_k \tag{3.2.6.2}$$

A simple example using the same sized data set from section 3.2.1 is demonstrated in Figure 3.4 and Figure 3.5, where b_k^* and C_k^* represent the updated values, which are refreshed every iteration. This means that after the first iteration d1

$$b_5^* = b_5 + C_5 b_6 \tag{3.2.6.3}$$

and subsequent to the second iteration d2

$$\begin{aligned} b_5^* &= b_5^* + C_5^* b_7^* \\ &= (b_5 + C_5 b_6) + (C_5 C_6)(b_7 + C_7 b_8) \end{aligned} \tag{3.2.6.4}$$

The important thing to note is that the value b_5^* after the first iteration d1 is not the same value as b_5^* after the second iteration d2.

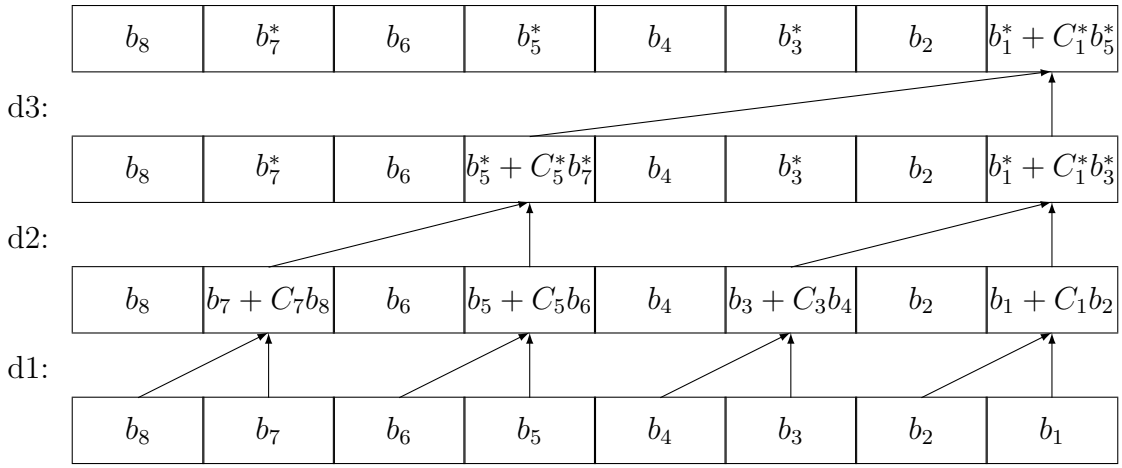

Figure 3.4: Up-Sweep Scan of b_k

The next step is to perform the down-sweep, which is identical to the up-sweep except that it runs in the reverse direction. There is something worth noting with regards to the down-sweep when referring to Figure 3.3, the down-sweep is initialized

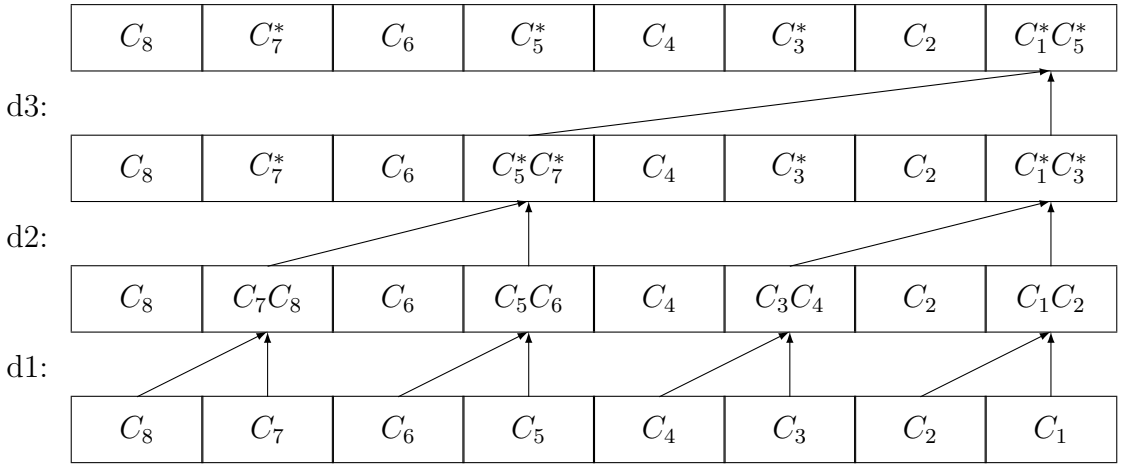

Figure 3.5: Up-Sweep Scan of C_k

by setting the last value equal to 0. This only applies if the scan algorithm is being applied to scalar values. In this case, vectors and matrices are being used, which consists of setting $b_1 = 0$ and $C_1 = I$, where I is the identity matrix. Always keep in mind that retrodiction is the pursuit of the estimate at time k given the estimates for all time greater than and including $k + 1$, which is why the first value in the scan algorithm shown in Figure 3.4 is b_8 and the last value is b_1.

Once the down-sweep is completed all of the values are shifted to the left by one index and the last index is set equal to the value calculated in the up-sweep, which means $b_1^* + C_1^* b_5^*$ and $C_1^* = C_1^* C_5^*$. It should be stated for clarity that b_1^* in Figure 3.6 is set equal to $b_3^* + C_3^* b_5^*$ after iteration d2.

At this point an observant reader might ask how the up and down-sweep detailed in figures 3.4 through 3.7 managed to solve for the variable a_k defined in (3.2.3.3)? The answer can be found by looking at a single computation from Figure 3.4, specifically

$$b_7 = b_7 + C_7 b_8 \tag{3.2.6.5}$$

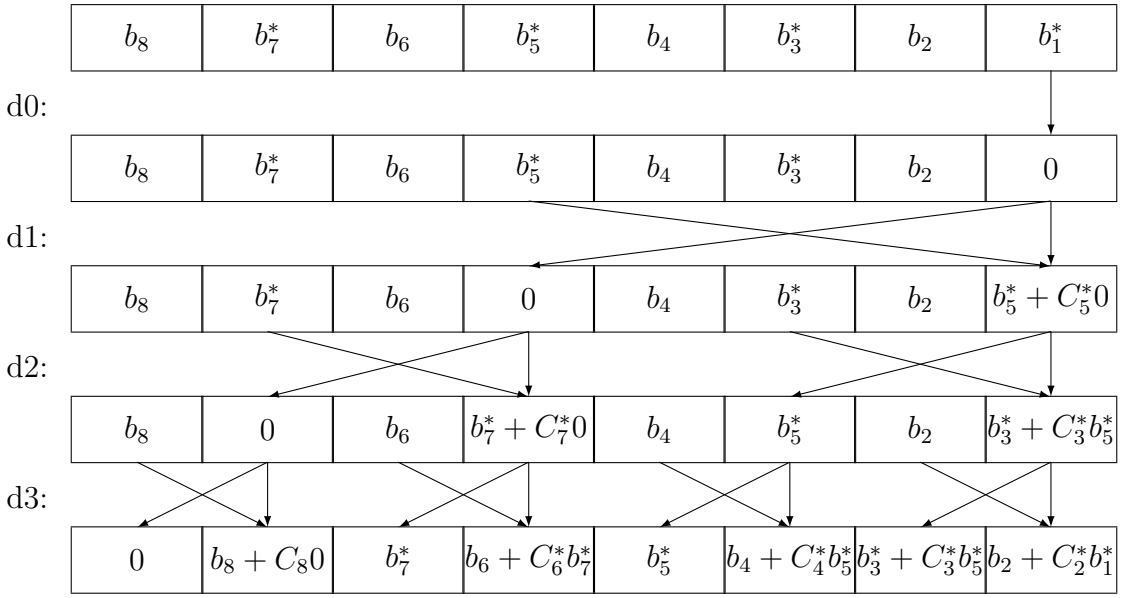

Figure 3.6: Down-Sweep Scan of b_k

Substituting (3.2.3.2) into the above equation results in

$$b_7 + C_7 b_8 = \hat{x}_{7|7} - C_7 F_7 \hat{x}_{7|7} + C_7(\hat{x}_{8|8} - C_8 F_8 \hat{x}_{8|8}) \tag{3.2.6.6}$$

Referring to equations (3.2.3.2) and (3.2.3.3), it is clear that

$$b_7 = \hat{x}_{7|7} - C_7 F_7 \hat{x}_{7|7} \tag{3.2.6.7}$$

and

$$a_7 x_8 = C_7(\hat{x}_{8|8} - C_8 F_8 \hat{x}_{8|8}) \tag{3.2.6.8}$$

This makes it obvious that

$$a_7 = C_7 \tag{3.2.6.9}$$

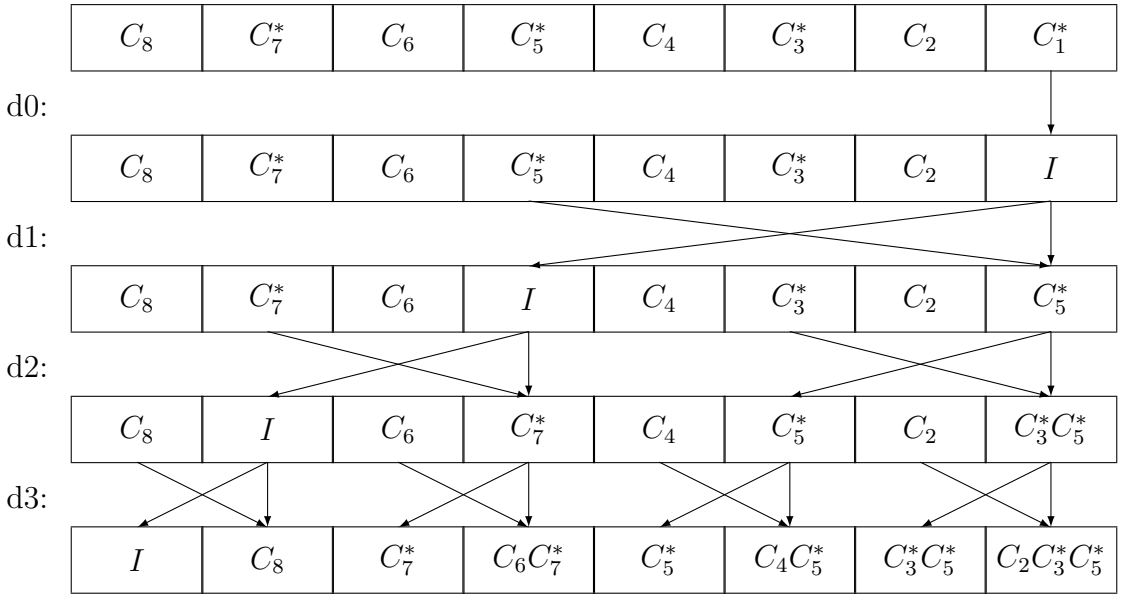

Figure 3.7: Down-Sweep Scan of C_k

while the following can be said about the estimated smoothed state

$$\begin{aligned} x_8 &= \hat{x}_{8|8} - C_8 F_8 \hat{x}_{8|8} \\ &= \hat{x}_{8|N} \end{aligned} \tag{3.2.6.10}$$

So the variable a_k is computed within the up and down-sweep, it has just been replaced by the variable C_k in figures 3.4 through 3.7. The fact that $a_k = C_k$ is no coincidence, it is the subtle beauty of the prefix sums algorithm. It is also a welcomed advantage since one would have to compute C_k in order to solve the Smoothing problem, yet instead of having to explicitly work out a_k as was the case for b_k, the scan provided it for free.

The last thing that requires attention is the inverse calculation of the matrices $(H_{k+1}P_{k+1|k+1}H'_{k+1} - R_k)^{-1}$ and $P_{k|k}^{-1}$ in (3.1.1.2) and (3.1.6), respectively. Since there are 2 state space variables and 2 dimensions, the size of these matrices is known

beforehand, which allows one to hard code the inverse calculation, thus circumventing the need for any overly complicated algorithm like LU decomposition or the Cholesky factorisation. The first matrix has dimensions 2×2, and can be computed using the formula outlined in (2.2.4.1). The estimated covariance matrix has dimensions 4×4, and its inverse is found using the adjugate matrix and the determinant [36]

$$A^{-1} = \frac{1}{\det(A)} \text{adj} A \tag{3.2.6.11}$$

Everything up until now has detailed all the ingredients needed to calculate the retrodiction of the Kalman filter using the prefix sums algorithm. But these are only half of the necessary steps that one would have to perform, there still remains the matter of the estimated covariance, which can be computed using the same procedure outlined in this section. There are, however, two additional steps that need to be addressed, that being the *vec* operation and the Kronecker product. Recall that the largest matrix was assumed to be 4×4 and performing the *vec* operator will transform this into a 16×1 vector, while the Kronecker product will create a 16×16 matrix. The first problem is that each block was allocated 4×4 threads, which cannot perform the necessary matrix-vector calculation from (3.2.4.2). One option is to use 4×4 threads and block operations, thus partitioning the 16×16 matrix into 16 sub matrices that can be placed into 16 distinct blocks. The problem with this is that one cannot use shared memory since blocks cannot arithmetically interact with one another, making it difficult to perform the product $(C_k \bigotimes C_k)vec(P_{k+1|N})$. A better option is to allocate 16×16 threads, but use 4×4 for all computations except those involving the *vec* and Kronecker operations. Once these transformations are completed all of the steps detailed in figures 3.4 to 3.7 can be easily applied.

As a result of the aforementioned dilemma, for the purposes of testing, the estimation covariance was not computed, solely the state estimation was calculated on both the CPU and the GPU. The reasoning behind this is due to the added necessity of having to solve for the estimation covariance matrices using the Kronecker product outlined in Appendix B, that would have forced the coding to be severely more complex than is needed in order to test the functionality of the parallelization of the Smoothing problem. Furthermore, the smoothed trajectory requires solely the state estimates, which is why the estimated covariance matrices can be ignored.

3.2.7 Simulation Results II: CPU vs GPU

As was the case in section 2.2.6, the following simulation was performed in two phases, one was performed on an i7 870 @ 2.93 GHz using an ANSI C compiler, while the second was executed on a NVIDIA GeForce GTX 570 graphics card using the CUDA runtime library. Like before, the serial algorithm of the retrodiction was written by this author using the C programming language so as to ensure that the code being executed was that which was outlined in [34]. The amount of data that was processed was increased from 64 to 524,288 in increments of 64×2^n, for $n = 1, ..., 13$. The corresponding runtime for both the CPU and the GPU was recorded and stored in Table 3.2 for quick referencing.

Data Size	CPU Time (s)	GPU Time (s)
64	0.001155	0.000372
128	0.005709	0.000621
256	0.009056	0.001009
512	0.030386	0.002039

1024	0.035425	0.004093
2048	0.069176	0.007815
4096	0.898497	0.015266
8192	1.01493	0.03052
16384	0.917155	0.061148
32768	1.093203	0.122052
65536	2.21806	0.244505
131072	4.376406	0.487737
262144	8.551049	0.974989
524288	17.162158	1.950584

Table 3.2: Retrodiction Simulation Results

Like Table 2.4, the results in Table 3.2 demonstrate how the GPU performs favorably and predictably, while the CPU is several times slower (refer to Figure 3.8). One noticeable difference between the results obtained in section 2.2.6 and those shown here are that the retrodiction algorithm was able to run an extra data set of size 524,288. The reason this was not done in section 2.2.6 was due to the fact that any data set size greater than 262,144 caused the device to crash. This crash was the consequence of utilizing too many registers per thread. Debugging this is outside the scope of this thesis and in no way undermines the simulation results.

The next thing worth mentioning is that there was greater consistency with the retrodiction algorithm when using the CPU, as can be seen in Figure 3.9. The unexplained behaviour of the CPU that was first observed in section 2.2.6 is less pronounced, albeit still present. The most interesting thing is how the shape of the graph

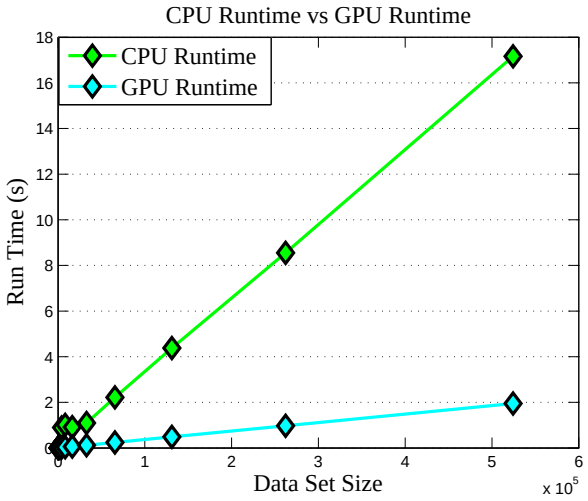

Figure 3.8: Retrodiction Runtime Comparison

for the CPU in Figure 2.2 is strikingly similar to that of Figure 3.9. The fact that two different algorithms performing two distinct sets of matrix/vector computations produced such analogous graphs suggests that whatever the problem is, it is not random and seems to be a systematic fault.

Another issue to consider in this section is that the retrodiction takes less time to compute than the prediction, which cab be observed when comparing Table 2.4 with Table 3.2. It seems that the Smoothing problem is about twice as fast as the Kalman filter. One might think that this is due to the fact that the calculations involving the smoothed estimation covariance were ignored, but that is not the sole reason. Recall from section 3.1.2 that in order to solve for $P_{k|k}$, the value of $P_{k+1|k}$ was calculated as well. However, once the Kalman filter computed $P_{k|k}$, this matrix was retained in memory and used later by the retrodiction algorithm. So the 9 matrix operations needed to solve for $P_{k+1|k}$ and the 3 accounting for $P_{k|k}$ were completely circumvented, which means that a total of 15 matrix/vector computations were performed. Also, the matrix transpose multiplication of $P_{k|k}F_k^T$ found in both (3.1.4) and (3.1.6) was

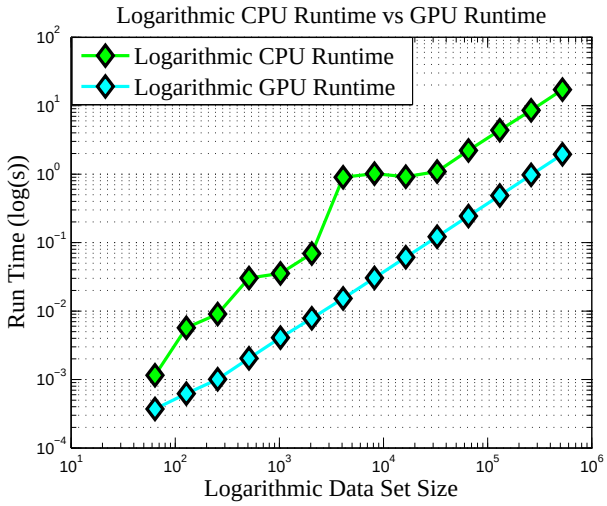

Figure 3.9: Logarithmic Retrodiction Runtime Comparison

computed only once since transpose matrix multiplication is performed in one step. Furthermore, $P_{k|k}F_k^T$ was calculated in (3.1.4) and then saved in memory and the result used again later in (3.1.6). This brings the total number of matrix/vector computations to 12. Subtract from this the 5 computations that were not executed when ignoring $P_{k|N}$ and the final total is 7 matrix/vector operations.

The only thing left to discuss in this section is the precision of the results. Like those produced in section 2.2.6, the results here were acquired using arrays of type `float`, for both the CPU and the GPU. Once again, the results did not demonstrate any kind of truncation or rounding off errors. The values computed for both the host and device were identical.

3.2.8 Simulation Results II: Block Size

Since the source code in section 3.2.6 relies heavily upon the number of blocks allocated, it makes sense to use this as a performance metric. This was the same tactic

seen in section 2.2.7, wherein the runtime for a varying number of blocks was catalogued and compared. Recall that in section 2.2.7 the code was executed for 1, 2, and 4 blocks. This does not make much sense here since the software outlined in section 3.2.6 was implemented using 64 blocks. Therefore, it seemed more appropriate to vary the number of blocks from 1 to 64, the results of which are shown in Table 3.3.

Data Size	1 Block	16 Blocks	64 Blocks
64	0.005899	0.000612	0.000372
128	0.011371	0.001396	0.000621
256	0.022945	0.00216	0.001009
512	0.045243	0.00413	0.002039
1024	0.090262	0.008267	0.004093
2048	0.180401	0.016376	0.007815
4096	0.360666	0.03288	0.015266
8192	0.721054	0.065386	0.03052
16384	1.442205	0.130914	0.061148

Table 3.3: Retrodiction Simulation Results

The first thing that stands out is how slow the GPU performs when only 1 block is being utilized. When compared to Table 3.2, it can be seen that the code is executed faster on the CPU than the GPU. This is the first time that the CPU has outperformed the GPU, and is a good lesson to remember. If one is not careful with the allocation of threads and blocks, the GPU can actually impede the runtime. This situation, however, only applies to very small block sizes, and quickly evaporates as the block size is increased to 16 and 64.

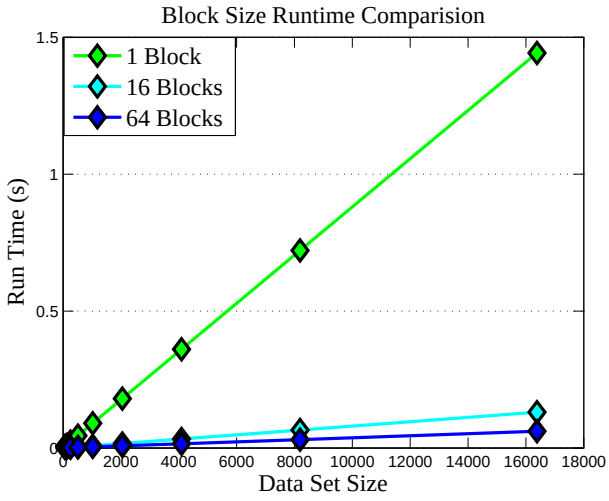

Figure 3.10: Retrodiction Block Size Comparison

The fact that the difference between utilizing 16 and 64 blocks only doubles the runtime is interesting because there are 4 times the number of blocks, yet only twice the improvement. This behaviour was not observed in Table 2.4, which displayed a runtime that was approximately halved when the block size was doubled, a pattern that seemed to suggest some linear relationship between the number of blocks and the runtime. Figure 3.10, however, shows that increasing the number of blocks can greatly affect performance, but does so in a non linear fashion. Its almost as if the number of blocks allocated will eventually saturate the performance, after which adding more blocks will contribute very little improvement.

3.2.9 Future Considerations II

It becomes apparent that as the design process evolves, there are numerous considerations to make when programming software for a GPU. Whether or not to use shared memory, how many threads need to be implemented, can these threads be divided

into a greater number of blocks, are just a few questions that have been addressed in the previous sections. Recall also that section 2.2.8 listed several other design goals that could help improve the performance of the GPU. Before delving into Visual Profiler, it seems wise to list the possible improvements discussed thus far.

1. Increase number of threads
    - The number of threads per block being utilized in the parallelization algorithm of the Smoothing problem is 16, this value could conceivably be increased to 256
2. Utilize global memory
    - The use of global memory in place of shared memory due to the data transfers from shared to global at the commencement of the up and down-sweep
3. Compute the estimation covariance matrix
    - Use the Kronecker product and the *vec* operator to compute the estimation covariance matrix in order to gauge the speed up

The whole point of trying to predict where performance enhancements on a GPU can be made is the result of refining one's skills so as to avoid pitfalls that might slow down the code, and seek out improvements that might speed up the code. Since writing software for a GPU is exponentially more complicated than that of a CPU, one needs to be cautious. The last thing that should happen is for the code to take several months to develop only to find out that its not possible to attain speed ups greater than a factor of 2 or 3. Yet, the situation becomes complicated when one doesn't know

exactly what to expect and is therefore incapable of determining beforehand whether or not the algorithm will perform admirably. This is the situation in which items 1 and 2 describe. There is no way of knowing how well or poorly the software will perform by altering the number of operational threads or using less shared memory in favor of global memory. At this point it really is just a matter of trial and error, which is a costly endeavour when writing GPU software. This is why the algorithm detailed in section 3.2.4 was never implemented. Having to write an extra kernel in order to perform the *vec* operation would have been an involved process that might or might not produce favorable results. But it is nonetheless, a novel approach to a growing field and worthy of mention.

As always, it is good practice to use any kind of tool capable of measuring performance metrics since optimizing the GPU for speed is the ultimate goal. Fortunately, NVIDIA provides this tool, which was already mentioned in section 2.2.8, called Visual Profiler. Therefore, the parallelization of the Smoothing problem was analysed and the following results produced.

1. Low memory copy and compute overlap
    - The percentage of time when memory copy is being performed in parallel with compute is low
2. Low memory copy throughput
    - The memory copies are not fully using the available host to device bandwidth
3. Low kernel concurrency

- The percentage of time when two kernels are being executed in parallel is low

4. Occupancy may be limited by block size
    - Occupancy can potentially be improved by increasing the number of threads per block
5. Occupancy may be limited by shared memory
    - Occupancy can potentially be improved by decreasing shared memory usage per block
6. Low multiprocessor occupancy
    - Low occupancy may limit utilization of the GPU's multiprocessors
7. Low global load efficiency
    - Global memory loads may have a poor access pattern, leading to inefficient use of global memory bandwidth
8. Low global memory store efficiency
    - Global memory stores may have a poor access pattern, leading to inefficient use of global memory bandwidth
9. Inefficient block size
    - Compute resources are being wasted because the number of threads per block is not a multiple of the warp size

10. High instruction replay overhead

    - A combination of global, shared, and local memory replays are causing significant instruction issue overhead

11. High global memory instruction replay overhead

    - Non-coalesced global memory accesses are causing significant instruction issue overhead

The first 3 items above are the same as those explained in section 2.2.8, where the reader is referred to for further information. Items 4 and 9 suggest that the number of threads be some multiple of 32 since that is the warp size. This ensures that the optimal number of threads are operational when they are launched. The next item 5 is something that was already discussed in the previous bullet listing, wherein the possibility that global memory might be better suited for the task instead of shared memory. Notice that items 4, 5 and 6 all mention problems areas related to occupancy. Occupancy is a metric used to determine how well the threads are accessing the registers. If only a few threads utilize a great many registers, then the occupancy is low, if the threads use just the right number of registers then the occupancy is optimized. As was discussed in section 2.2.5, this is an involved topic that goes to the heart of understanding the advanced nature of a GPU's architecture. The three items 7, 8 and 11 were already addressed in section 2.2.8. Item 10 is one of the more complicated and vague issues since it suggests that there may be a problem with the use of memory. It states that a better execution of either global, shared, local, textured, constant or registers is advised. The only hint towards any resolution is the one provided in section 3.2.6 that states the use of global memory might be

better suited to meet the needs of this particular algorithm than shared memory.

Chapter 4

Conclusion

Surely performing block computations will enhance the performance of a simple tracker, enabling it to monitor a greater number of targets that it was incapable of doing before GPUs were available, but that is merely a stepping stone to the larger picture. What about all of the publications concerning data fusion [5] [6] [23]? How does the ability of a tracker to monitor numerous targets simultaneously affect the trackers ability to monitor multiple sensors simultaneously? From the perspective of a GPU the two are not so different, which means that the details outlined in chapter 2 can be easily adopted to the field of data fusion. So not only can a GPU be used to perform faster matrix computations once the data has been acquired, it can be used to acquire the data faster. This means that the architecture of a graphical processing unit can be utilized at various stages of the data processing. So the next inevitable question is how many different ways, and in how many different places can a GPU improve the whole system of monitoring targets? At this point no one knows, which is what makes this thesis so compelling, it sets the stage for what's to come. Deriving parallel algorithms for MHT, PDA, JPDA and Alpha-Beta filters has to start

somewhere and is too intractable a task to perform in one sitting. It is something that will require several years and hundreds of publications before it is brought to a level of maturity that can be utilized reliably in industry. Yet, the chore has to start somewhere and at some point, and starting with the Kalman filter was a judicious choice since it paves the way for a parallel IMM filter, while planting the seed for those who might consider parallelizing the extended and unscented Kalman filters [8] [37].

Yet, exploiting the functionality of a GPU by performing numerous operations concurrently is a limited endeavour that will eventually saturate, after all even a GPU has limits. Its not so much the speed up that should concern a designer, but the architecture. Differing architecture has differing applications, some optimized for certain scenarios and not for others. This is exactly why there are several variants of processors such as microcontrollers, digital signal processors and field programmable gate arrays. Each one has its place and functionality. This is the whole point of deriving a parallel algorithm for the retrodiction of the Kalman filter. Being smart about how to exploit the hardware allows the software to go the extra mile. Structuring the Smoothing problem so that it fits into the prefix sums algorithm is equivalent to compressing data into a smaller storage space before it is transferred, after all why perform block operations on a serial algorithm when it is possible to perform those same block operations on a parallel algorithm? So the parallelization of the Smoothing problem is an additional advantage to the already pre-existing advantage that a GPU provides when performing matrix/vector calculations. The retrodiction described in [34] could have easily been computed using the same block partitioning

technique outlined in chapter 2, but it would have amounted to block operations being performed in a recursive manner. Instead, block operations were performed in a parallel manner, resulting in a refined efficiency.

So the only question that remains is what does all this mean for the future? There is no doubt that the industry places greater demand on the tracking community as a whole to monitor more targets with ever increasing precision. Greater precision means better algorithms which usually entails more complicated math. Just think of the IMM filter, which can switch between models, or the MHT filter that keeps track of multiple scenarios, both of these are modern filters that require a greater number of computations than their predecessors the Alpha-Beta or Kalman filter. And what of optical tracking methods, the pixel count in cameras is increasing almost everyday, this requires more processing power in order to perform operations like edge detection [38] and feature extraction [33] that are fundamental in tracking. The only way to solve these challenges is with GPUs. But blindly using threads and blocks in hopes that speed ups will occur is unrealistic. A strategic approach has to be adopted, one wherein the computations are optimized along with the algorithms.

Appendix A

Block Matrix

A matrix can be partitioned into sub matrices referred to as blocks in the following manner, suppose

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

then the partitioned matrix can be defined as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where

$$A_{11} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}$$

$$A_{12} = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix}$$

$$A_{22} = \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix}$$

This idea can be extended to matrix multiplication by performing the operation on the sub matrices. For example, let

$$B = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 5 & 5 & 6 & 6 \\ 7 & 7 & 8 & 8 \end{bmatrix}$$

If B is partitioned into the same sized sub matrices as A, then

$$\begin{aligned} C &= AB \\ &= \begin{bmatrix} A_{11}B_{11} & A_{12}B_{12} \\ A_{21}B_{21} & A_{22}B_{22} \end{bmatrix} \\ &= \begin{bmatrix} 7 & 7 & 22 & 22 \\ 23 & 23 & 46 & 46 \\ 115 & 115 & 162 & 162 \\ 163 & 163 & 218 & 218 \end{bmatrix} \end{aligned}$$

Appendix B

Vectorization

The vectorization performs the linear transformation of a matrix into vector form. This means that a matrix A, can be converted into a column vector. Suppose

$$A = \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix}$$

where A is a $m \times n$ matrix with columns $a_1 \dots a_n$, then the *vec* operation is defined as

$$vec(A) = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

and the dimensions of the resulting vector are $mn \times 1$.

An extension of the *vec* operation is the Kronecker product that utilizes the results shown above in order to perform matrix multiplication. The Kronecker product is

defined as follows

$$vec(AXB) = (B^T \otimes A)vec(X)$$

The operator $\otimes$ denotes the Kronecker product which if defined for an $m \times n$ matrix A and a $p \times q$ matrix B, results in

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ a_{21}B & a_{22}B & \dots & a_{2n}B \\ \vdots & \vdots & \dots & \vdots \\ a_{m1}B & a_{m2}B & \dots & a_{mn}B \end{bmatrix}$$

where the resulting matrix has dimensions $mp \times nq$. For example, let

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

and let

$$B = \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix}$$

Then the following holds true

$$A \otimes B = \begin{bmatrix} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 48 \end{bmatrix}$$

Appendix C

GeForce GTX 570 Specifications

The following specifications were obtained through the use of the CUDA device query.

Description	Value
Device	GeForce GTX 570
CUDA Driver Version	5.0
CUDA Capability	2.0
Total amount of global memory	1280 MB
(15) Multiprocessors × (32) CUDA Cores/MP	480 CUDA Cores
GPU Clock rate	1484 MHz
Memory Clock rate	1900 MHz
Memory Bus Width	320-bit
L2 Cache Size	655360 bytes
Max Textured Dimension Size (x,y,z)	1D=65536, 2D=65536,65536, 3D=2048,2048,2048

Max Layered Texture Size (dim × layers)	1D=16384×2048, 2D=(16384,16384)×2048
Total amount of constant memory	65536
Total amount of shared memory per block	49152 bytes
Total amount of registers available per block	32768
Warp size	32
Maximum number of threads per multiprocessor	1536
Maximum number of threads per block	1024
Maximum block size	$1024 \times 1024 \times 64$
Maximum grid size	$65535 \times 65535 \times 65535$
Maximum memory pitch	2147483647 bytes
Texture alignment	512
Concurrency copy and kernel execution	Yes with 1 copy engine
Runtime limit on kernel	Yes
Integrated GPU sharing Host Memory	No
Support host page-locked memory mapping	Yes
Alignment requirement for Surfaces	Yes
Device has ECC support	Disabled
Device supports Unified Addressing (UVA)	No
Deice PCI Bus ID/PCI location ID	1/0

Table C.1: CUDA Device Query (Runtime API)

Bibliography

[1] S. Arulampalam, N. Gordon, B. Ristic. *"Beyond Kalman Filter: Particle Filters for Tracking Applications"*. Artech House, Boston, MA, USA, January 2004.

[2] Y. Bar-Shalom, H. Blom. "The Interacting Multiple Model Algorithm for Systems with Markovian Switching Coefficients". *IEEE Transactions on Automatic Control*, vol. 33, no. 8, pp. 780–783, August 1988.

[3] Y. Bar-Shalom, F. Daum, J. Huang. "The Probabilistic Data Association Filter". *IEEE Control Systems Magazine*, vol. 29, no. 6, pp. 82–100, December 2009.

[4] Y. Bar-Shalom, K. Chang. "Joint Probabilistic Data Association for Multitarget Tracking with Possibly Unresolved Measurements and Maneuver". *IEEE Transactions on Automatic Control*, vol. 29, no. 7, pp. "585–594", 1984.

[5] Y. Bar-Shalom, I. Kadar, T. Kirubarajan, K. R. Pattipati. "Large Scale Ground Target Tracking With Single and Multiple MTI Sensors". *Multitarget-Multisensor Tracking Applications and Advances III*, 2000.

[6] Y. Bar-Shalom, X. R. Li. *"Multitarget-Multisensor Tracking: Principles and Techniques"*. YBS Publishing, Storrs, USA, 1995.

[7] Y. Bar-Shalom, T. Kirubarajan, X. R. Li. *Estimation with Applications to Tracking and Navigation: Theory Algorithms and Software*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[8] G. Bishop, G. Welch. "An Introduction to the Kalman Filter". *Department of Computer Science, University of North Carolina*, 2001.

[9] S. Blackman. "Multiple Hypothesis Tracking For Multiple Target Tracking". *IEEE Transactions on Aerospace and Electronic Systems*, vol. 19, no. 1, pp. 5–18, 2003.

[10] G. E. Blelloch. "Prefix Sums and Their Applications". *School of Computer Science, Carnegie Mellon University*, (CMU-CS-90-190), November 1990.

[11] C. B. Chang, K. P. Dunn, D. Willner. "Kalman Filter Algorithms for a Multi-Sensor System". *IEEE Conference on Decision and Control including the 15th Symposium on Adaptive Processes*, pp. 570–574, December 1976.

[12] Y. Chang, B. Huang, M. Huang S. Wei. "Accelerating the Kalman Filter on a GPU". *IEEE 17th International Conference on Parallel and Distributed Systems*, pp. 1016–1020, 2011.

[13] T. Cormen, C. Leiserson, R. Rivest, C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 3rd edition, 2009.

[14] O. E. Drummond. "Feature, Attribute, and Classification Aided Target Tracking". *SPIE Proceedings, Signal and Data Processing of Small Targets*, vol. 4473, pp. 542–558, November 2001.

[15] H. Eves. *Elementary Matrix Theory*. Dover Publications Inc., Mineola, NY, USA, 1980.

[16] J. B. Fraleigh, V. J. Katz. *A First Course in Abstract Algebra*. Addison-Wesley, London, ON, Canada, 7th edition, 2003.

[17] H. R. Hashemipour, A. J. Laub, S. Roy. "Decentralized Structures for Parallel Kalman Filtering". *IEEE Transactions on Automatic Control*, vol. 33, no. 1, pp. 88–94, January 1988.

[18] R. E. Kalman. "A New Approach to Linear Filtering and Prediction Problems". *Transactions of the ASMEJournal of Basic Engineering*, vol. 82, pp. 35–45, March 1960.

[19] J. M. Kessenich, B. M. Licea-Kane, G. Sellers, D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL(R), Version 2*. Addison-Wesley, 5th edition, 2005.

[20] B. Khailany M. Bauer, H. Cook. "CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization". *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2011.

[21] T. Kirubarajan, T. Lang, M. McDonald, N. Nandakumaran, S. Sutharsan, R. Tharmarasa. "Interacting Multiple Model Forward Filtering and Backward Smoothing for Maneuvering Target Tracking". *Signal and Data Processing of Small Targets*, vol. 7445, 2009.

[22] W. Koch. "GMTI-tracking and information fusion for ground surveillance". *Signal and Data Processing of Small Targets*, vol. 4473, pp. 381, 2001.

[23] J. Koller, M. Ulmke. "Data Fusion for Ground Moving Target Tracking". *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, vol. 11, no. 4, pp. 261–270, May 2006.

[24] B. C. Levy, A. S. Willsky, A. H. Tewfik. "A New Parallel Smoothing Algorithm". *IEEE Conference on Descicion and Control*, vol. 25, pp. 933–937, December 1986.

[25] B. C. Levy, A. S. Willsky, A. H. Tewfik. "Parallel Smoothing". *Systems and Control*, vol. 14, no. 3, pp. 253–259, March 1988.

[26] X. R. Li. "The PDF of Nearest Neighbor Measurement and a Probalistic Nearest Neighbor filter for Tracking in Clutter". *IEEE Conference on Decision and Control*, vol. 1, pp. 918–923, December 1993.

[27] J. R. Magnus, H. Neudecker. *Matrix Differential Calculus with Applications in Statistics and Econometrics*. Wiley, March 1999.

[28] NVIDIA Corporation. "Compute Visual Profiler: User Giude". (DU-05162-001_v02), October 2010. url: http://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[29] NVIDIA Corporation. "NVIDIA CUDA: Programming Guide". (Version 1.0), 2010. url: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[30] NVIDIA Corporation. "NVIDIA CUDA: Reference Manual". (Version 3.2 Beta), August 2010. url: http://docs.nvidia.com/cuda/cuda-runtime-api/index.html.

[31] NVIDIA Corporation. "Cuda C Best Practices Guide". (DG-05603-001_v4.0), May 2011. url: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[32] A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill, New York, NY, USA, 4th edition, 1984.

[33] C. Poullis, U. Neumann, S. You. "Linear Feature Extraction Using Perceptual Grouping and Graph-Cuts". *15th ACM International Symposium on Geographic Information Systems*, pp. 64, November 2007.

[34] H. E. Rauch, C. T. Striebel, F. Tung. "Maximum Likelihood Estimates of Linear Dynamic Systems". *AIAA Journal*, vol. 3, no. 8, pp. 1445–1450, August 1965.

[35] J. Stam. "Maximizing GPU Efficiency in Extreme Throughput Applications". *GPU Technology Conference*, October 2009.

[36] G. Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge, Wellesley, MA, USA, 3rd edition, 2005.

[37] R. van der Menve, E. A. Wan. "The Unscented Kalman Filter for Nonlinear Estimation". *IEEE Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000*, pp. 153–158, October 2000.

[38] X. Wang. "Gating Techniques for Maneuvering Target in Clutter". *IEEE Transactions on Aerospace and Electronic Systems*, vol. 38, no. 3, pp. 1087–1097, July 2002.