Co-processor Offloading Applied to Passive Coherent Location with Doppler and Bearing Data

Joe Milburn

February 4, 2010

Supervised by: Professor Michael Inggs

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in Engineering in the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

Signature of Author . . . .
.

Cape Town

February 4, 2010

Abstract

This project dealt with the acceleration of an aircraft tracking algorithm using a ClearSpeed mathematical co-processor. The algorithm is based on non-linear differential correction (also known as the Gauss-Newton method) and uses Doppler and bearing data from a Passive Coherent Location (PCL) radar system. A PCL radar uses a network of receivers to track targets through their back-scatter from existing Continuous Wave (CW) transmissions, such as broadcast TV or radio. The lack of an active transmitter in a PCL system results in relatively low procurement, operation and maintenance costs. This is of particular advantage for airports in third world countries, many of which do not have radar assisted air traffic control.

Differential correction combines data from a number of radar receivers using the minimum variance rule. New data is incorporated as it is received in order to continually correct the model in an optimal fashion. The model produced is a 10th degree polynomial which describes the trajectory of the aircraft in a Cartesian co-ordinate system.

The objectives of the dissertation were to identify the computationally intensive areas of the algorithm, then accelerate performance by offloading computation to the co-processor. The co-processor used was the ClearSpeed Advance X620 accelerator board, which fits into the PCI-X slot of a conventional computing platform. It is claimed that the Advance board range uses the fastest and most power efficient double-precision 64-bit floating point processors in the world. Application acceleration is achieved via two methods, parallelization and hardware optimized library routines. An investigation into acceleration of the Gauss-Newton algorithm with ClearSpeed was deemed worthwhile as the algorithm makes use of linear algebra routines supported by ClearSpeed, as well as computationally intensive double precision arithmetic.

Profiling of the pre-existing implementation of the algorithm (written by Dr. Richard Lord in IDL) revealed that the most computationally expensive areas are arithmetic operations that calculate partial derivatives of the observation functions, and high dimension double precision matrix manipulations. The Gauss-Newton algorithm was successfully implemented in C, with accuracy results that compared favourably with the original IDL implementation. The performance of the C version (with no ClearSpeed acceleration) was 1.38 times faster than the IDL implementation, mainly due to the efficiency of the hardware tuned ATLAS BLAS library.

It was found that the sizes of the matrices involved in the multiplications are not a good fit for direct acceleration via the provided ClearSpeed library functions. Further investigation concluded that a moderate speedup could be attained through parallelization of one of the matrix multiplications due to the sparse and data redundant nature of the input matrices. The accuracy results verified the correct functionality of the ClearSpeed accelerated algorithm, however the results showed that the estimates produced were an order of magnitude less accurate than the IDL version. This can be attributed to the differences in accuracy of the card side matrix multiplication and the IDL / BLAS library routines. A 2.22 times increase in performance was achieved (over the C implementation) through the co-processor offloading.

Given the amount of programming effort required to achieve this moderate speedup, it can be concluded that the Gauss-Newton algorithm is not a good fit for ClearSpeed assisted acceleration.

Acknowledgments

First and foremost I thank Professor Michael Inggs for his infinite patience, invaluable guidance, feedback, and words of encouragement and advice throughout the course of this project. I will always be grateful for his understanding, knowledge and professionalism. He sets a fine example not just as an engineer and academic, but also as a person.

I would also particularly like to acknowledge and thank Dr. Norman Morrison for his ground-breaking work on aircraft tracking with the Gauss-Newton method that paved the way for this project. His technical knowledge and expertise are second to none, and without his guidance I would not have known where to start with this project. For allowing me access to his books on the subject and for always opening his door to me to answer my many questions I am eternally grateful.

I acknowledge Dr. Richard Lord's hard work on the IDL simulator program that implemented Dr. Morrison's tracking algorithm, which provided an ample starting point and reference. Without his work the project would have been infinitely more difficult if not impossible.

I would like to acknowledge the PCL research group at UCL who continue to research in parallel with Professor Inggs on PCL tracking. I thank in particular Professor Chris Baker and Dr. Carl Woodbridge for their input on their visits to UCT.

I thank everyone at the RRSG (past and present) for assisting, encouraging and showing me the ropes along the way. Particularly Regine Lord for her organizational and administrative skills, her contribution and value during her time as a member of the group cannot be understated. Others that spring to mind are Aadil Volkwin, Gunther Lange, Roufurd Julie, Lance Williams, Marc Brooker, Jonathan Ward, Fabien Blangy, Simon Winberg, Samuel Ginsberg and Dr. Yoann Paichard.

I thank everyone at the CHPC for allowing me to use their facilities and aiding me with their technical knowledge along the way, and also being patient with me and my research progress. In particular I thank Albert Gazendam for his co-supervisory role early on in the project, Dr. Khomotso Kganyago for giving me a push and for the opportunity to work with the ClearSpeed cards, Dr. Jeff Chen for all his help and encouragement and Jeremy Main for all his technical assistance, he was always friendly and helpful. I also thank Dr. Happy Sithole, Omar Ismail (IBM), Eric Mbele, and Sticks Mabakane. I thank those in the ACE group for their company in the lab and for helping me weather the research storm. They include among others Mike Aitken, Michael Gorven, Jane Hewitson, Nick Thorne, Andrew Woods and Jean-Paul de Conceicao. Thanks to J-P for helping me to get the code running again when I'd long since forgotten how it worked, and for taking over the project.

I would like to acknowledge the ClearSpeed support and technical team for their assistance in answering my questions and helping me with various technical issues I had along the way. In particular I thank David McCormick, Jamie Packer and Tim Bainbridge.

Finally I thank my family and friends for all the support. I particularly thank my father for funding my research and believing in me through it all. To the many others that I have omitted to mention here that helped me along the way throughout these three years, my deepest thanks.

Contents

1 Introduction **1**

1.1 Problem statement . . . 1

1.2 Background . . . 1

1.3 Objectives . . . 2

1.4 Plan of development . . . 3

1.4.1 Outline . . . 3

1.4.2 Description of hardware . . . 3

1.4.3 The Gauss-Newton tracking algorithm . . . 5

1.4.4 Profiling of the IDL implementation of the Gauss-Newton algorithm . . . 6

1.4.5 Implementation of Gauss-Newton in C . . . 11

1.4.6 Acceleration of the Gauss-Newton algorithm with ClearSpeed . . . 14

1.4.7 Conclusions . . . 21

2 Description of hardware **23**

2.1 Co-processor assisted acceleration . . . 23

2.2 The ClearSpeed Advance X620 board . . . 23

2.2.1 The CSX600 parallel processor . . . 24

2.3 The ClearSpeed software environment . . . 26

2.3.1 Acceleration with the CSXL library . . . 27

2.3.2 Acceleration through parallelization . . . 30

3 The Gauss-Newton tracking algorithm **33**

3.1 Definitions . . . 33

3.2 The observation equations . . . 33

3.3 T matrix calculation . . . 35

3.3.1 The polynomial transition matrix and the perturbation vector . . . 35

3.3.2 The observation sensitivity matrix 35
3.3.3 The total observation matrix calculation 37
3.4 The Minimum Variance Rule 38
3.4.1 The residuals 38
3.4.2 Derivation of the minimum variance algorithm 39
3.5 The iterative Gauss-Newton algorithm 40
3.5.1 Initialization of the nominal state vector 41
3.5.2 The stopping rule for Gauss-Newton iteration 41

4 IDL implementation profiling 43
4.1 Radar data generation 44
4.2 Master Control Algorithm profiling 44
4.3 Gauss-Newton method profiling 47

5 Implementation of Gauss-Newton tracking in C 50
5.1 Description of Implementation 50
5.2 C implementation accuracy testing 52
5.2.1 Single precision accuracy testing 53
5.3 C implementation profiling 54

6 Acceleration of the Gauss-Newton algorithm with ClearSpeed 58
6.1 Design of ClearSpeed assisted implementation 58
6.1.1 Proposed acceleration of the total observation matrix calculation 60
6.1.2 Prediction of performance of accelerated observation matrix calculation 64
6.1.3 Prediction of performance of CSXL matrix multiplications 66
6.2 ClearSpeed assisted Gauss-Newton algorithm 67
6.2.1 Description of ClearSpeed assisted implementation 67
6.2.2 Verification of ClearSpeed assisted implementation 69
6.2.3 ClearSpeed accelerated algorithm profiling 71

7 Conclusions 74
7.1 Limitations of implementation 74
7.2 Comments on programming with ClearSpeed 74
7.3 Results of acceleration 75
7.4 Future work 75

List of Figures

1.1 PCL for air traffic control with the Gauss-Newton method . . . 2
1.2 ClearSpeed Advance X620 accelerator block diagram . . . 4
1.3 ClearSpeed software architecture and CSXL . . . 4
1.4 Block diagram of pre-existing PCL simulator used to test Gauss-Newton algorithm . . . 7
1.5 Graphical output from IDL simulator . . . 8
1.6 Graph showing profile of MCA IDL implementation . . . 9
1.7 Graph illustrating profiling of the Gauss-Newton algorithm in IDL . . . 10
1.8 Block diagram of C implementation of MCA with accuracy testing and profiling . . . 11
1.9 Graphs of comparative C and IDL implementation accuracy . . . 12
1.10 Gauss-Newton C implementation profile . . . 13
1.11 Comparative C and IDL profiling graph . . . 14
1.12 Intended ClearSpeed accelerated Gauss Newton implementation flow of control . . . 15
1.13 CSXL vs ATLAS matrix multiplication execution times . . . 16
1.14 ClearSpeed accelerated Gauss Newton flow chart . . . 17
1.15 ClearSpeed accelerated accuracy results . . . 19
1.16 ClearSpeed accelerated host processing profile . . . 20
1.17 ClearSpeed accelerated card side processing profile . . . 20
1.18 ClearSpeed accelerated performance increase graph . . . 21

2.1 ClearSpeed Advance X620 architecture block diagram . . . 24
2.2 ClearSpeed CSX600 processor architecture block diagram . . . 25
2.3 The CSX600 Multi-Threaded Array Processor (MTAP) . . . 26
2.4 ClearSpeed software architecture . . . 27
2.5 Host application calling BLAS functions . . . 29
2.6 CSAPI driver library interaction time-line . . . 30

3.1 The residuals of the observation and fitted observation vectors 38
3.2 Sum of weighted squared residuals as a surface 40
3.3 1st degree current estimate EMP algorithm 41

4.1 IDL simulator block diagram 43
4.2 Master Control Algorithm flow diagram for the IDL implementation 46
4.3 Profiling of MCA showing percentage execution time 47
4.4 Graph illustrating Gauss-Newton IDL implementation profiling 48

5.1 Block diagram of C implementation of algorithm with accuracy testing and profiling 50
5.2 C implementation of Gauss-Newton algorithm 51
5.3 Graphs showing accuracy of C implementation 53
5.4 C Implementation profiling 54
5.5 Execution times for observation matrix C implementation 55
5.6 Execution times for covariance matrix C version 55
5.7 Distribution of matrix multiplication execution times for C version 56
5.8 C vs IDL implementation profile comparison graph 57

6.1 Identification of areas of computation as candidates for ClearSpeed acceleration 59
6.2 Intended ClearSpeed accelerated Gauss Newton implementation flow of control 60
6.3 Structure of the observation sensitivity matrix 62
6.4 Structure of the state vector transition matrix 63
6.5 Total observation matrix computation on one ClearSpeed processing element 64
6.6 CSXL vs ATLAS matrix multiplication execution times 67
6.7 ClearSpeed accelerated Gauss Newton flow chart 68
6.8 ClearSpeed accelerated Gauss-Newton MCA 69
6.9 ClearSpeed accelerated accuracy results 70
6.10 Accelerated tracking output visualization with IDL simulator program 71
6.11 ClearSpeed accelerated host processing profile 72
6.12 ClearSpeed accelerated card side processing profile 72
6.13 ClearSpeed accelerated performance increase graph 73

List of Tables

1.1 Base platform specification 6
1.2 Comparative stopping rule statistics of C implementation and ClearSpeed assisted algorithm 18

2.1 Semaphore properties 31

4.1 Base platform specification 44

6.1 Sensitivity matrix calculation poly data requirement per processing element 63
6.2 Transfer rates for total observation matrix calculation data structures 65
6.3 Number of instructions for T matrix calculation 65
6.4 Predicted times for T matrix computation instructions 66
6.5 Matrix sizes involved in Gauss-Newton filtering 67
6.6 Comparative stopping rule statistics of C implementation and ClearSpeed assisted algorithm 70

7.1 IDL implementation profiling summary 83
7.2 Unassisted C implementation profiling summary 84
7.3 ClearSpeed accelerated implementation profiling summary 85
7.4 IDL accuracy result summary 86
7.5 Unassisted C accuracy result summary 86
7.6 ClearSpeed accelerated accuracy result summary 86

Nomenclature

ALU - Arithmetic Logic Unit

API - Application Programming Interface

ATC - Air Traffic Control

ATLAS - Automatically Tuned Linear Algebra Software

BLAS - Basic Linear Algebra Subprograms

CCBR - ClearConnect Busbridge port

CHPC - Centre for High Performance Computing

DST - Department of Science and Technology

CPU - Central Processing Unit

CSIR - Centre for Scientific and Industrial Research

CW - Continuous Wave

DGEMM - Double precision GEneral Matrix Multiplication

DMA - Direct Memory Access

DSP - Digital Signal Processing

EMP - Expanding Memory Polynomial

FFT - Fast Fourier Transform

FPGA - Field Programmable Gate Array

GPGPU - General Purpose computing on Graphics Processing Units

GSU - Global Semaphore Unit

HDP - Host interface Debug Port

HPC - High Performance Computing

IDL - Interactive Data Language

ISU - Interrupt Semaphore Unit

LAPACK - Linear Algebra PACKage

MAC - Multiply-Accumulate unit

MTAP - Multi-Threaded Array Processor

NaN - Not a Number

NoC - Network on a Chip

PCI - Peripheral Component Interconnect

PCI-X - PCI eXtended

PCL - Passive Coherent Location

SDRAM - Synchronous Dynamic Random Access Memory

SoC - System on a Chip

SRAM - Static Random Access Memory

TSC - Thread Sequence Controller

UCT - University of Cape Town

Chapter 1

Introduction

1.1 Problem statement

This dissertation dealt with the acceleration of an aircraft tracking algorithm using a ClearSpeed mathematical co-processor. The algorithm is based on non-linear differential correction and uses Doppler and bearing data from a Passive Coherent Location (PCL) radar system.

1.2 Background

Africa has one of the largest aircraft accident rates per flying hour in the world. According to statistics compiled in 2006 Africa had the highest five year average fatal airliner accident rate in the world, and was also the only region whose accident rate was not decreasing [5]. One factor that contributes to unsafe flying conditions in many African and other third world countries, is the lack of infrastructure and equipment required for Aircraft Traffic Control (ATC). Pilots landing at such airports have to rely on their own eyes to survey the runway from the air before landing. The lack of an active transmitter and resulting cost saving makes PCL radar attractive not just for countries that would otherwise be unable to afford radar systems, but for future ATC implementations in general (see Figure 1.1).

The tracking algorithm under consideration was developed by Dr Norman Morrison, Dr Richard Lord and Professor Michael Inggs of UCT, as part of the ongoing investigation into PCL radar systems jointly undertaken by the University of Cape Town and the University College London. The algorithm is based on the idea that as computer processing becomes cheaper and faster, expensive hardware can be replaced by mathematical computation. The algorithm processes the Doppler and bearing data to produce a state vector estimation of the target. This state vector specifies position, velocity and higher order derivatives of position in a three dimensional Cartesian coordinate system. The state vector is updated based on new data, discarding older. The system uses non-linear differential correction, also known as the Gauss-Newton method.[1] The Gauss-Newton algorithm involves matrix inversions and multiplications which are computationally expensive. A diagram of a possible PCL traffic control system is shown in Figure 1.1.

[1]Note that the terms differential correction and Gauss-Newton are used interchangeably in this document, as are the terms filtering and tracking

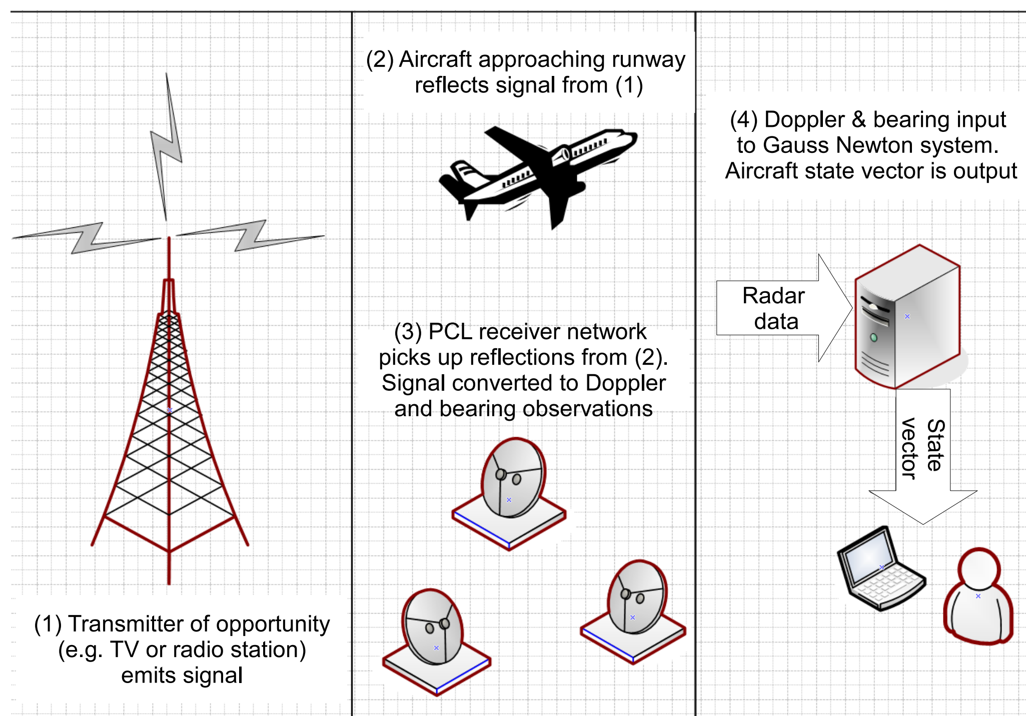

Figure 1.1: PCL for ATC with the Gauss-Newton method. A PCL radar uses a network of receivers to track targets through their back-scatter from existing Continuous Wave (CW) transmissions, such as broadcast TV or radio. The lack of an active transmitter in a PCL system results in relatively low procurement, operation and maintenance costs.

In the course of the investigation into possible hardware platforms for the implementation, a ClearSpeed Advance X620 co-processor board became available for use at the Centre for High Performance Computing (CHPC). The CHPC is an initiative of the Department of Science and Technology (DST) and is managed by the Centre for Scientific and Industrial Research (CSIR) and the University of Cape Town (UCT) in South Africa. The co-processor board is designed to accelerate computationally intensive double precision floating point processing. ClearSpeed provides a set of libraries that provide acceleration of commonly used linear algebra routines. An investigation into acceleration of the Gauss-Newton algorithm with ClearSpeed was deemed worthwhile as the algorithm makes use of linear algebra routines supported by ClearSpeed, as well as computationally intensive double precision arithmetic.

1.3 Objectives

The objectives of the dissertation were to:

- Identify the most computationally intensive areas of the algorithm.
- Port the algorithm to a hardware platform with a High Performance Computing (HPC) co-processor.
- Accelerate the implementation through offloading of computation to the co-processor.
- Test the co-processor accelerated version of the algorithm for accuracy.
- Test the performance of the algorithm on the HPC system compared to the base system.

1.4 Plan of development

1.4.1 Outline

The following chapters of this dissertation include discussions of:

- Application acceleration with ClearSpeed from a hardware and software perspective.
- The mathematical and filtering concepts involved in the Gauss-Newton algorithm.
- The performance profile of the pre-existing tracking algorithm implementation (written in IDL).
- The C implementation of the algorithm on the base platform, including profiling and accuracy testing.
- Computation offloading of the implementation with the ClearSpeed co-processor.
- Performance testing on the ClearSpeed accelerated platform.

1.4.2 Description of hardware

Chapter 2 contains a description of the ClearSpeed co-processor hardware and software, as well as some of the programming concepts involved.

The traditional method of increasing computing performance by increasing clock speed and transistor density of a von Neumann architecture CPU is now facing a brick wall limit in terms of the amount of energy that a single microprocessor die can dissipate [1]. Therefore innovations in computer architecture and the replacement of fast, monolithic, single processor dies with multiple slower processors operating in parallel must substitute for the traditional method [1].

One approach to increasing compute performance is the use of co-processor accelerator boards. Such boards can offer dramatic power and space benefits over other forms of HPC while offering impressive performance increases over conventional serial computing architectures, if the algorithm in question is well suited to the co-processor hardware architecture.

The co-processor used for this project was the ClearSpeed Advance X620 accelerator board, which fits into the PCI-X slot of a conventional computing platform. It is claimed that the Advance board range uses the fastest and most power efficient double-precision 64-bit floating point processors in the world. A block diagram of the board is shown in Figure 1.2. One CSX600 processor contains 96 execution cores or processing elements (PEs) .

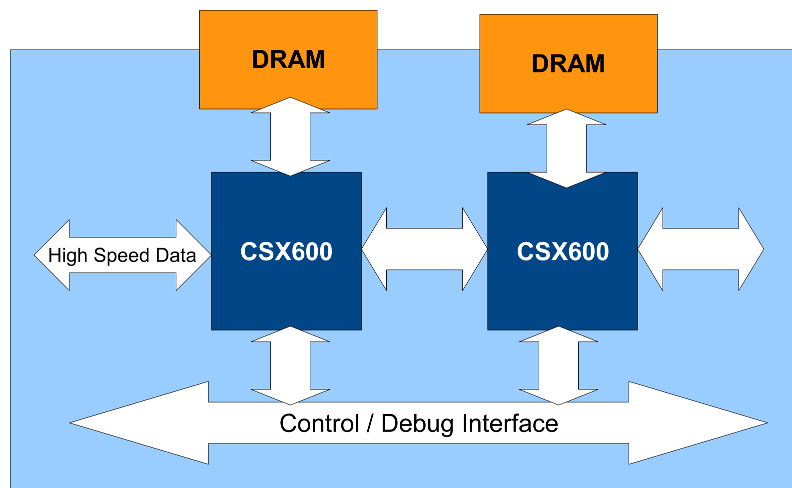

Figure 1.2: ClearSpeed Advance X620 accelerator block diagram. The X620 consists of two CSX600 embedded parallel processors communicating via the ClearConnect busbridge ports, 1GB of local DDR2-400 SD-RAM and an FPGA which implements the host interface.[13]

The ClearSpeed software environment consists of the runtime environment (including device drivers), the CSXL and CSDFT application acceleration libraries, and the Software Development Kit (SDK). The relationship between them is shown in Figure 1.3.

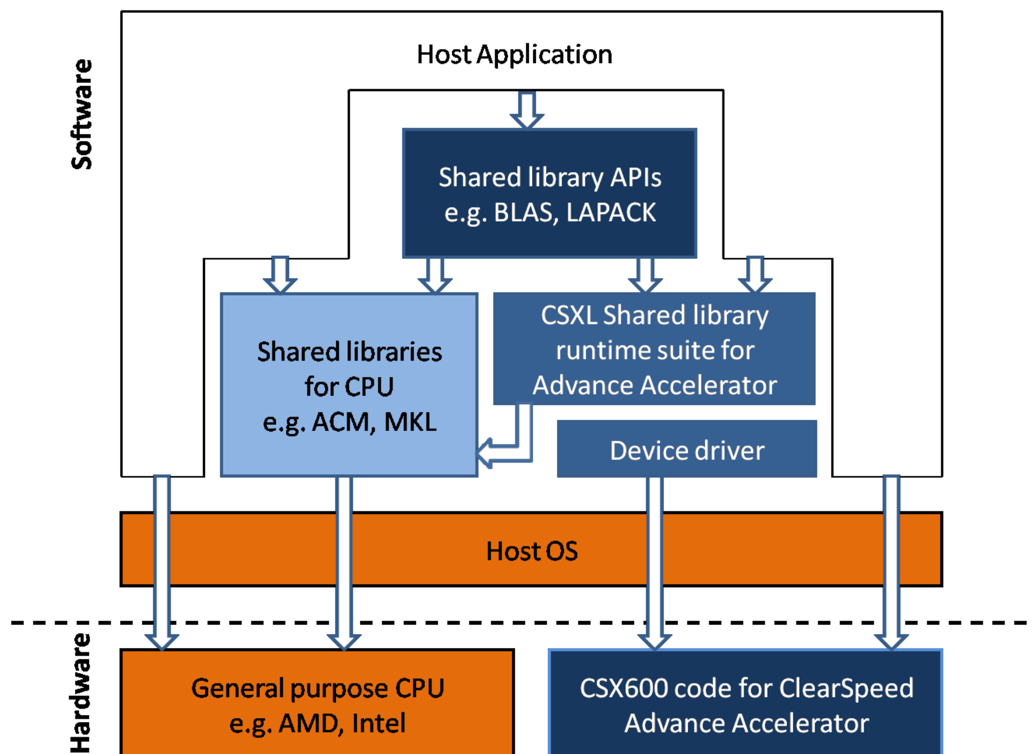

Figure 1.3: ClearSpeed software architecture and CSXL[13]

Application acceleration with the ClearSpeed Advance X620 is achieved via two methods, parallelization and through hardware optimized versions of various linear algebra routines.

Conclusions

In order to be candidates for parallelization, applications need to be data parallel, have a high computation to data movement ratio and small working data sets [8]. Keeping within the memory requirements of the CSX600 requires some care, and performance is vastly improved if it is possible to overlap computation and data transfer [10]. This goes for communication between the host and co-processor as well as for data transfer between the various tiers of on-board memory.

An application is also a candidate for acceleration if a high percentage of application execution time is taken up by linear algebra routines supported by ClearSpeed's CSXL library (or the CSDFT library which performs various FFT functions). When the host application calls a library routine, the runtime environment uses heuristics to determine whether to run that routine on the host or the Advance board [14]. Performance benefits are only significant when the problem size is a good fit for the ClearSpeed hardware.

1.4.3 The Gauss-Newton tracking algorithm

Chapter 3 contains an outline of some of the statistical and mathematical principles involved in Gauss-Newton filtering.

The model developed through non-linear differential correction is a polynomial of degree 10. The input to the algorithm is an observation vector containing radar measurements, as well as an initial estimate of the state vector. Let the total observation vector[2] $Y_{(n)}$ be a matrix containing radar observations for every time instant up to t_n. Each of the observation variables can then be expressed as a known function of the state vector. Let the true total state vector up to time t_n be $X_{(n)}$. Formula 1.1[3] shows the relationship between $X_{(n)}$ and $Y_{(n)}$, where $N_{(n)}$ is a noise vector.

$$Y_{(n)} = F(X_{(n)}) + N_{(n)} \tag{1.1}$$

Assume we have a nominal total state vector for the target up to time t_n, $\bar{X}_{(n)}$. $\bar{X}_{(n)}$ is close to $X_{(n)}$ in the sense of $\delta X_{(n)}$, as shown by formula 1.2 [3].

$$\delta X_{(n)} = X_{(n)} - \bar{X}_{(n)} \tag{1.2}$$

$\delta X_{(n)}$ is a vector of small numbers called the perturbation vector.

We can use the nominal total state vector $\bar{X}_{(n)}$ to generate a simulated observation vector, using equation 1.3 [3].

$$\bar{Y}_{(n)} = F(\bar{X}_{(n)}) \tag{1.3}$$

The difference between the actual and simulated observation vectors is the observation perturbation vector, and can be developed using formula 1.4 [3].

$$\delta Y_{(n)} = Y_{(n)} - \bar{Y}_{(n)} = F(X_{(n)}) - F(\bar{X}_{(n)}) + N_{(n)} \tag{1.4}$$

If we Taylor series expand each line of vectors we end up with equation 1.5[3].

$$\delta Y_{(n)} = T_{(n)} \delta X_n + N_{(n)} \tag{1.5}$$

[2] The notation $X_n$shall be used to represent the vector X valid for time instant t_n, whereas $X_{(n)}$ shall represent the total set of vectors from time t_{n-L} up to time t_n. Where L is the memory length of the system.

$T_{(n)}$ is a matrix derived from the known partial derivatives of $F(X_{(n)})$ with respect to the elements of the state vector, evaluated at $\bar{X}_{(n)}$, and is known as the total observation matrix. This last equation is an overdetermined system of linear equations[3]. From the minimum variance rule we can develop equation 1.6 [3].

$$\delta X_n = (T_{(n)}^T R_{(n)}^{-1} T_{(n)})^{-1} T_{(n)}^T R_{(n)}^{-1} \delta Y_{(n)} \tag{1.6}$$

$R_{(n)}$ is the covariance matrix of the total error vector $N_{(n)}$, constructed from the variances of the noise distributions of the observations at each time instance. The result obtained above is used to give a new approximation to the state vector, as shown by equation 1.7[3].

$$(\bar{X}_n)_{new} = (\bar{X}_n)_{old} + \delta X_n \tag{1.7}$$

The state vector $(\bar{X}_n)_{new}$ is then translated to all time instances using a state vector transition matrix $\Phi_{(n)}$.[4] $(\bar{X}_{(n)})_{old}$ is the complete state vector used in the current iteration, and $(\bar{X}_{(n)})_{new}$ becomes its replacement in the next iteration. A stopping rule or "goodness of fit" criteria is used to determine when to cease iterating differential correction. The rule uses the average value of the perturbation vector δX_n, and RMS value of the observation residuals (difference between simulated and actual Doppler and bearings), as well as a maximum number of Gauss-Newton iterations. The first two parameters can be used to determine reliably whether the filter is in a state of error/covariance matrix consistency and therefore whether a reasonable estimate has been produced [3]. Once the desired threshold is reached the first two terms of each dimension of $(\bar{X}_{(n)})_{final}$ gives the position and velocity estimate in 3d space.

Conclusions

Morrison [3]argues that the Gauss-Newton algorithm not only offers a legitimate alternative to conventional Kalman and Swerling filtering techniques, but offers significant benefit under the correct circumstances. The computationally intensive non-recursive Gauss-Newton filtering technique can be used for aircraft tracking in the modern era due to the massive advances in computing power since the advent of Kalman and Swerling filters in the 50s and 60s [3].

1.4.4 Profiling of the IDL implementation of the Gauss-Newton algorithm

Chapter 4 describes the profiling of the existing tracking algorithm implementation, which was written and tested with a radar simulator program by Dr Richard Lord in the IDL programming language [2].

The method used was to profile the implementation was to time the execution of functional blocks of code through calls to the system clock. The platform specifications are shown in Table 1.1.

Table 1.1: Base platform specification

CPU Family	Intel(R) Xeon(TM)
CPU Clock Speed	4 x 3.00GHz dual core (x86_64) with 2MB cache
Memory	2GB
Storage	2x 300GB SATA
Co-processor Interface	PCI-X 2.0 133 MHz
Operating System	SuSE Linux Enterprise Server 9 (SLES9)

[3] An overdetermined system of linear equations is a system with more equations than unknowns

[4] Note the distinction between $X_{bar,(n)}$and $X_{bar,n}$: The notation $*_{(n)}$ denotes multiple data samples for a single observation instant (all samples for the observation instant occurring at time n), while$*_n$ implies a single data sample for a single moment in time (time n).

A block diagram of the radar simulator is shown in Figure 1.4. Note that the radar data generator used in the simulator is fairly simplistic, and the Gauss-Newton implementation under investigation has yet to be tested under more realistic conditions. What is of main concern here is the tracking algorithm block, and that will therefore be the focus of this discussion. However a brief description of the radar data generator is included here to provide a clear understanding of the flow of data into the tracking algorithm.

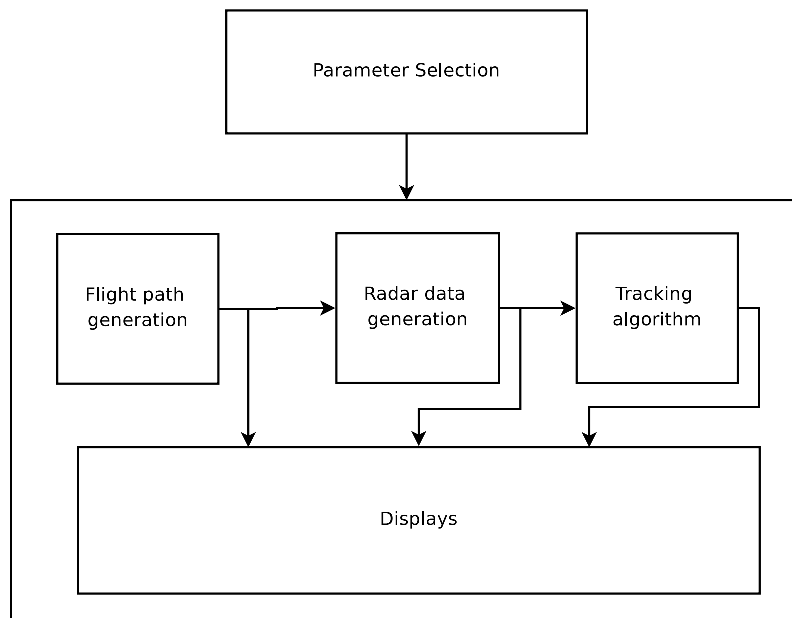

Figure 1.4: Block diagram of pre-existing PCL simulator, as written by Dr Richard Lord in the IDL programming language, which was used to test Gauss-Newton algorithm[3]

Radar data generation

The integration time "τ" for the cross-correlation function of the radar generator was 0.25s, resulting in 4 observation samples per second. Each sample consisted of a Doppler and Bearing reading from each of 8 receivers. The first set of observation data was sent to the tracking algorithm after 20seconds, and thus consisted of 80 samples. Subsequent cycles, or observation instances, occurred every 5seconds, and consisted of 80 samples (20 new observations and the last 60 from the previous cycle).

Tracking algorithm

The tracking algorithm can be split into three components: the master control algorithm (MCA), track initialization and the Gauss-Newton algorithm. The MCA controls the flow of data through the other stages. Track initialization is used to provide a partial, rough estimate of the state vector of the target aircraft using bearing observations. This estimate consists of position and velocity in the *X/Y* plane, and is fed into the Gauss-Newton algorithm, along with the associated Doppler and bearing observations. The Gauss-Newton algorithm iterates a number of times, each time outputting an increasingly accurate estimate of the target's state vector, until an accuracy threshold or "goodness of fit" criteria is reached. The state vector developed by the algorithm describes the flightpath of the aircraft using polynomials in three dimensional space.

A 10th degree polynomial was used for this implementation. The simulator uses the 1st two values of the polynomial to specify the position and velocity. Graphs of simulator output as well as receiver layout in the *X/Y plane* can be seen in Figure 1.5.

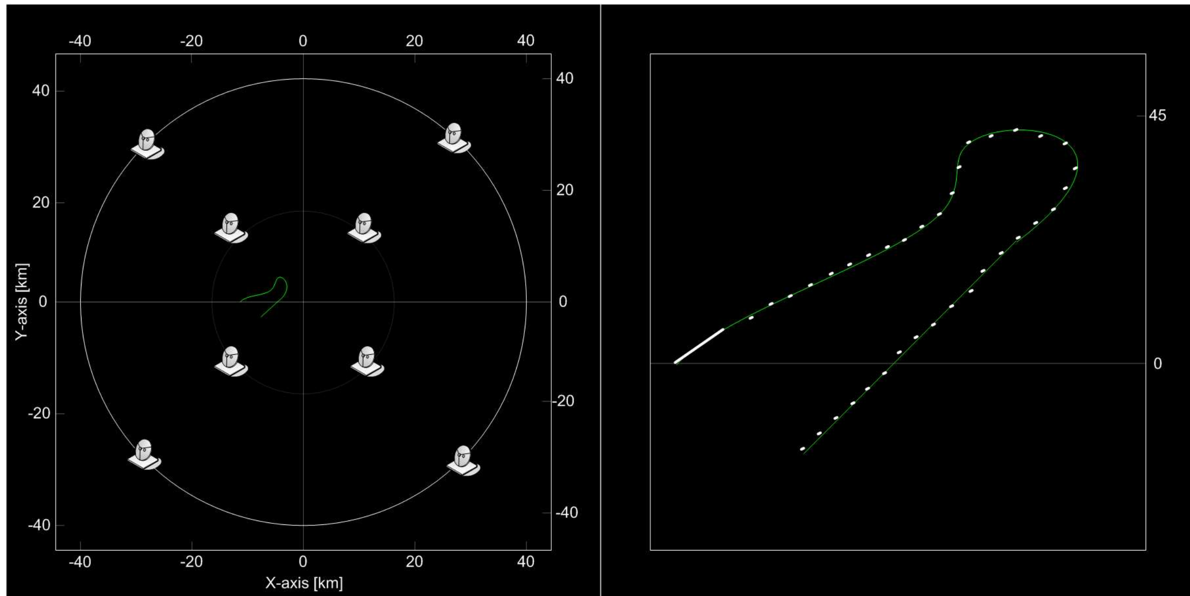

Figure 1.5: Graphs output by the pre-existing IDL simulator program showing receiver layout and tracking of landing passenger jet in X/Y plane[2]

Master Control Algorithm profiling

The flow of control of the MCA for one simulation run is as follows:

1. Initialize arrays, including the transition matrices $\Phi_{(n)}$ for the state vectors.
2. Initialize the target track.
3. Run the Gauss-Newton algorithm to produce $X_{final(n)}$.
4. Repeat parts 2 and 3 for all remaining state vector updates (observation instances).

In profiling the MCA times were averaged over 10 simulations, each with a different flightpath as input. Each simulation consists of 37 observation instances, therefore times for the main functional blocks are averaged over 370 readings. The results of the MCA profiling are shown in bar chart of Figure 1.6. As expected the Gauss-Newton algorithm accounts for the majority (about 94%) of the execution time.

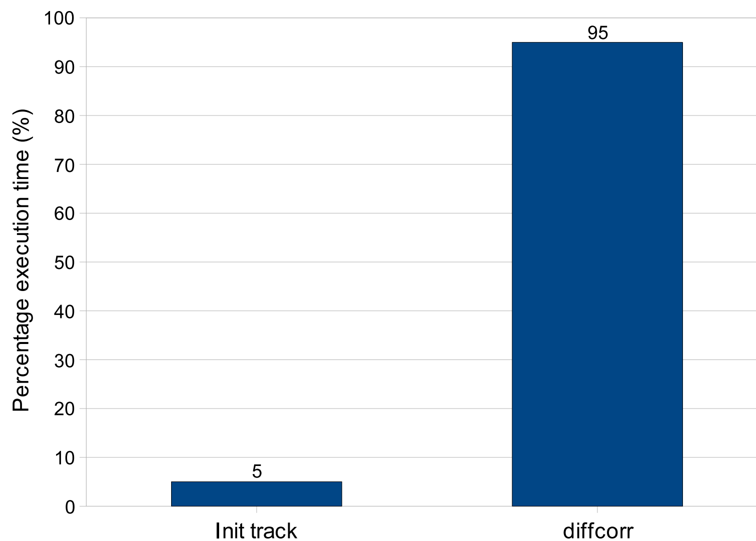

Figure 1.6: Profiling of the IDL implementation of the MCA showing percentage execution time. As expected the Gauss-Newton algorithm dominates the execution profile

Gauss-Newton method profiling

The operation of the Gauss-Newton method is described here:

1. Initialize arrays and parameters.
2. Iterate differential correction. Each differential correction iteration executes as follows:
   (a) Assemble the total simulated observation vector $\bar{Y}_{(n)}$ (using $\bar{X}_{(n)}$), and the observation perturbation vector $\delta Y_{(n)}$ ($\delta Y_{(n)} = Y_{(n)} - \bar{Y}_{(n)}$) as well as the RMS residuals of the observations.
   (b) Assemble the total observation matrix $T_{(n)}$. This is done in a loop that iterates over the data samples of the observation instance. Let the iterator i represent a single time instance. The stages of each iteration are:
      i. Calculate the sensitivity matrix $M(X_i)$ using the partial derivatives of the observation equations with respect to the elements of the state vector evaluated at $\bar{X}_i$.
      ii. Calculate a few rows of the total observation matrix $T_{(n)}$ by multiplying the sensitivity matrix by the transition matrix for this sample, Φ_i.
   (c) Transpose $T_{(n)}$ then multiply by $R^{-1}_{(n)}$ giving $T^T_{(n)}R^{-1}_{(n)}$.
   (d) Generate covariance matrix by computing $T^T_{(n)}R^{-1}_{(n)}T_{(n)}$, then inverting it to form $(T^T_{(n)}R^{-1}_{(n)}T_{(n)})^{-1}$.
   (e) Compute $\delta X_n = (T^T_{(n)}R^{-1}_{(n)}T_{(n)})^{-1}T^T_{(n)}R^{-1}_{(n)}\delta Y_{(n)}$.
   (f) Generate new state vector estimate for the most recent time instant: $(\bar{X}_n)_{new} = (\bar{X}_n)_{old} + \delta X_n$then build $(\bar{X}_{(n)})_{new}$ using the transition matrices$\Phi_{(n)}$and $(\bar{X}_{(n)})_{new}$
   (g) The algorithm then tests for convergence (or goodness of fit) using the RMS residuals, average of δX_n, and number of iterations that have occurred[5]. If the test is negative differential correction iterates again.

[5]The maximum number of iterations used in profiling was 50. This maximum has been found to be sufficient [citation], and indeed was never reached during the course of the profiling.

Conclusions

The results of the Gauss-Newton algorithm profiling are shown in Figure 1.7. The main areas of computation are:

- Observation perturbation vector loop. This is labeled *dY* in the figure and accounts for about 9% of the execution time.
- Total observation matrix loop:
  - Partial derivatives calculation for sensitivity matrix. Labeled *partials*, accounts for 26.5%.
  - Matrix multiplication to calculate $T_{(n)}$ rows. Labeled *T rows*, accounts for 27.7%. This includes matrix transpose and scaling to calculate $T^T_{(n)}R^{-1}_{(n)}$, which accounts for 6%.
- Matrix multiplication and inversion to form covariance matrix. Labeled *CovMat*, accounts for 12%.
- Matrix multiplications in perturbation vector calculation. Labeled *dX*, accounts for 21%.
- Retrodiction of state vector estimate to all time instances. Labeled *retrodict*, accounts for 2.79%.

These cumulatively account for 96% of the Gauss-Newton algorithm execution time.

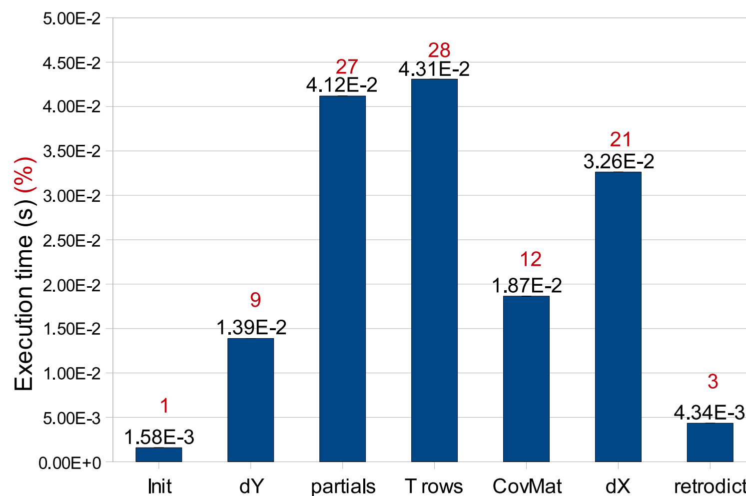

Figure 1.7: Graph illustrating IDL implementation of Gauss-Newton profiling. The observation perturbation vector loop is labeled *dY*. The partial derivatives calculation for the sensitivity matrix is labeled *partials*. The multiplications of the sensitivity matrices with the transition matrices is labeled *Trows*. This *Trows* execution block includes the observation matrix transpose and scaling with the error vector covariance matrix. Calculation of the covariance matrix $(T^T_{(n)}\, R^{-1}_{(n)}\, T_{(n)})^{-1}$ is labeled *covMat*. The multiplication of the covariance matrix $T^T_{(n)}\, R^{-1}_{(n)}$ and the observation observation perturbation vector to form the state vector perturbation vector is labeled *dX*. The addition of the perturbation vector to the previous state vector estimate (or nominal state vector), and the retrodiction of the new state vector estimate to all time instances using the transition matrix is labeled *retrodict*.

The profiling of the IDL implementation reveal that the most computationally expensive areas of the algorithm are the arithmetic operations in calculating the partial derivatives of the observation functions, and the high dimension matrix multiplications. The partial derivative calculations consist of a large concentration of double precision arithmetic, repeated in a loop.

1.4.5 Implementation of Gauss-Newton in C

Chapter 5 outlines the methodology used to port the Gauss-Newton filtering algorithm to the C programming language and the results of profiling and accuracy testing.

The MCA is illustrated by the block diagram in Figure 1.8.

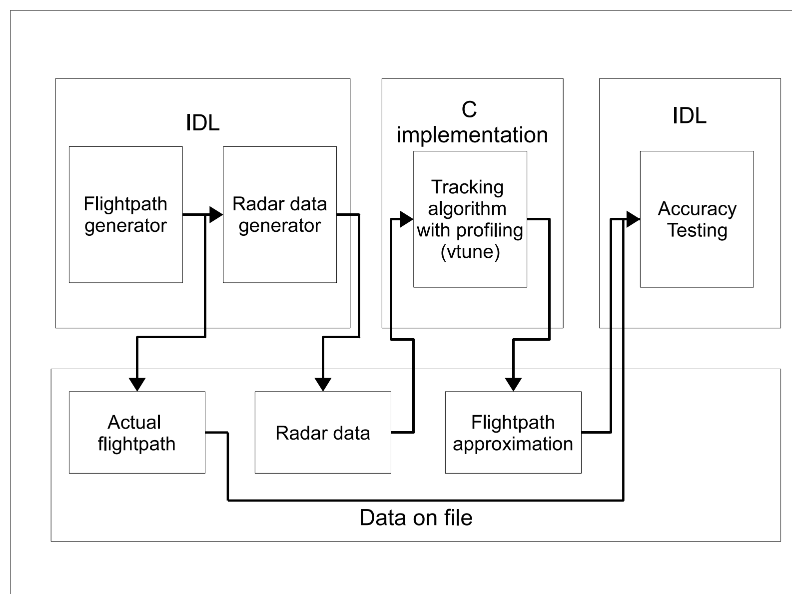

Figure 1.8: Block diagram of C implementation of MCA with accuracy testing and profiling. The flightpath and radar data generator from the IDL simulator were used to write the flight path and radar observations to file. This information was then read by the C code which implemented Gauss-Newton filtering to develop an approximation to the flightpath. The profiling was done using Intel's VTune Performance Analyzer.

C implementation accuracy testing

The flightpath approximation from C was compared with the actual flightpath generated by the IDL simulator using a simple IDL program. The accuracy was tested for 60 simulations, each with a different flightpath. Each simulation consisted of 37 observation instances, thus the accuracy data is taken over 2220 executions of the algorithm. The accuracy of the position and velocity output of the unaccelerated C implementation is illustrated by Figure 1.9. These accuracy values compare favourably with those of the original IDL Gauss-Newton implementation.

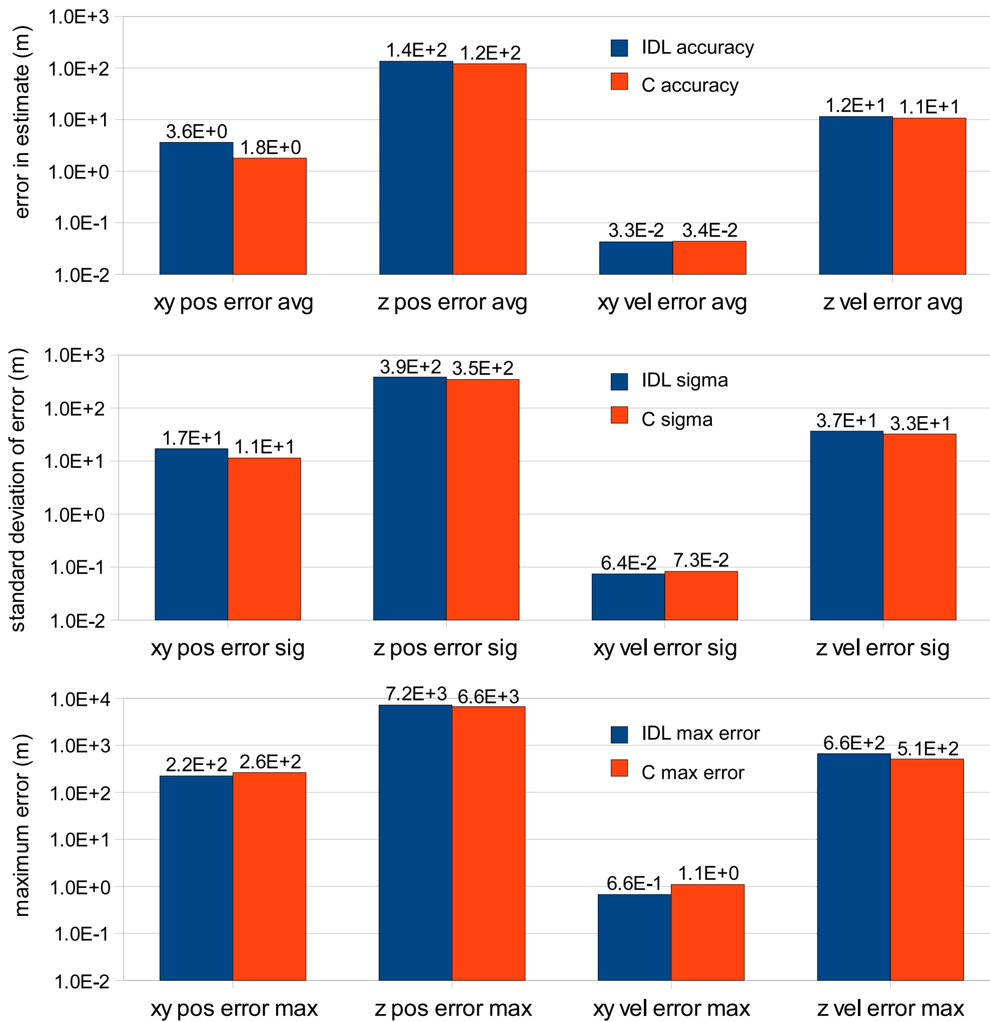

Figure 1.9: Graphs of comparative C and IDL implementation accuracy. The variables shown are position and velocity in the X/Y plane, as well as altitude and its rate of change. The average error over 2220 executions is shown in the top graph, followed by the standard deviation of these errors in the middle graph. Finally the maximum error of each of the state vector variables is shown in the lower graph. As is evident from the graphs the accuracy of the C implementation compares favourably with the IDL version.

Single precision accuracy testing

A version of the C implementation using single instead of double precision for the data structures was also tested. This would have been of benefit when offloading computation to the ClearSpeed board, both in terms of data transfer overhead and the limited memory (particularly poly memory) available on the card. Unfortunately single precision accuracy was found to be insufficient for the Gauss-Newton algorithm. The results of computation in the single precision version were progressively inaccurate as the errors propagated, the final results being unusable.

C implementation profiling

The C code was profiled using Intel's VTune Performance Analyzer. Results of the profiling are shown in Figure 1.10. Take note that the observation perturbation vector and total observation matrix loops were combined, the resulting execution block is labeled *T Matrix* in Figure 1.10.

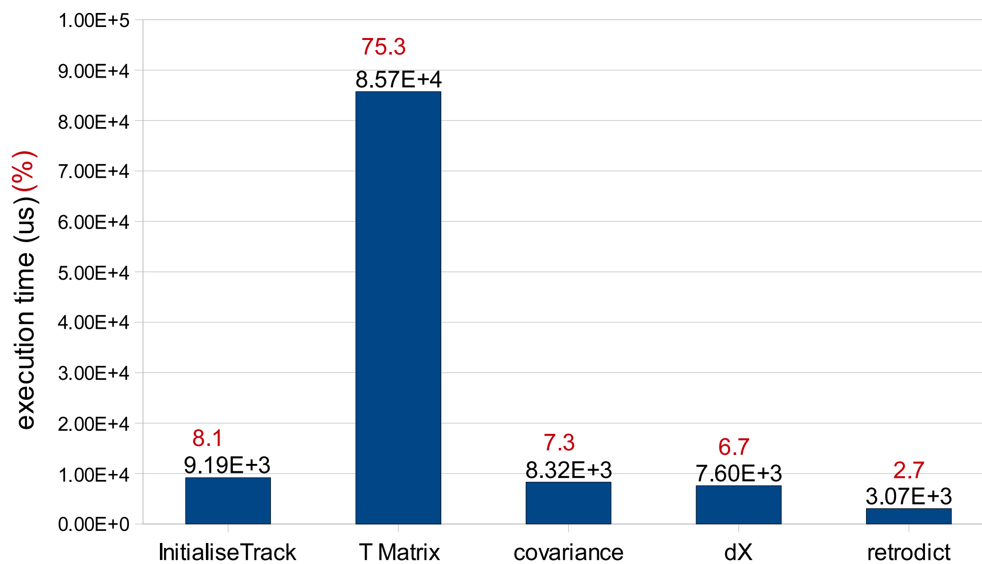

Figure 1.10: Gauss-Newton C implementation profile

Comparing the percentage execution times of C to IDL shows that in C matrix multiplication was relatively inexpensive. This can be attributed to the optimized matrix multiplication routines used in the C code, which make use of the ATLAS BLAS library. ATLAS libraries are tuned specifically for optimal performance on the host platform at build time. The C and IDL execution times are compared in Figure 1.11.

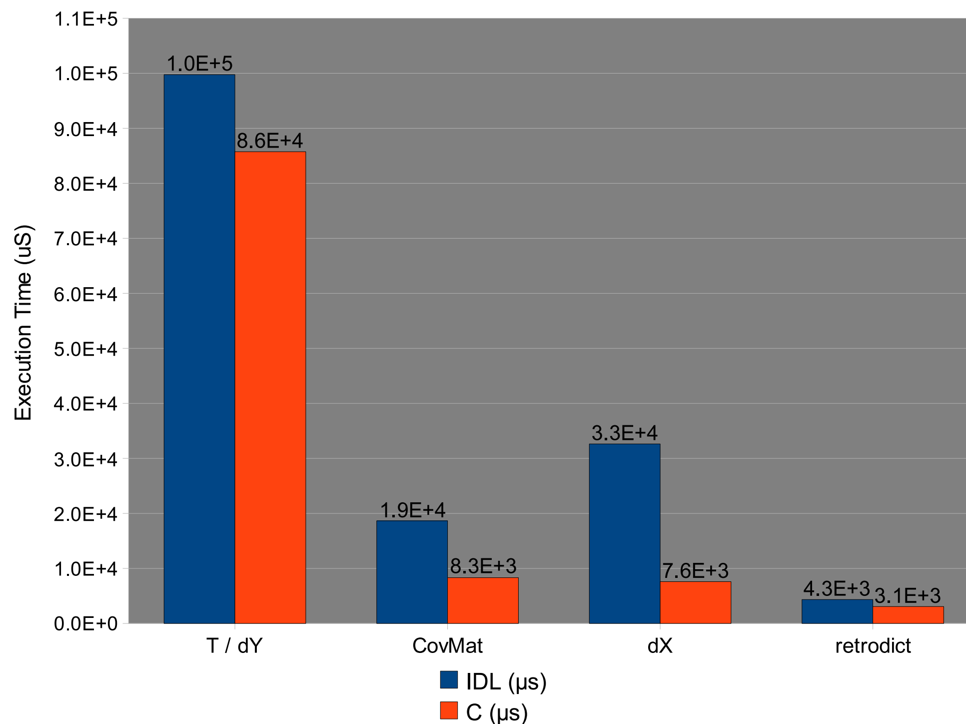

Figure 1.11: Comparative C and IDL profiling graph. Matrix multiplication is relatively inexpensive in the C implementation because of the use of the ATLAS BLAS library which is optimally tuned for performance on the host platform at build time.

Conclusions

The Gauss-Newton algorithm was successfully implemented, with accuracy results that compared favourably with the original IDL implementation. The performance of the C version was 1.38 times faster than the IDL implementation, mainly due to the efficiency of the hardware tuned ATLAS BLAS library.

1.4.6 Acceleration of the Gauss-Newton algorithm with ClearSpeed

Chapter 6 contains an investigation of co-processor offloading for Gauss-Newton tracking, as well as a description of the final ClearSpeed accelerated implementation.

Design

The design phase consisted of developing a flow chart showing the intended distribution of processing between the host and the accelerator. Estimates of the execution time of the accelerated sections of code were then made, including predictions of the overheads of both communication between host and accelerator and data transfers internal to the accelerator. The sections of code identified as candidates for ClearSpeed acceleration were those involving loops with no data dependency between iterations, and the high dimension double precision matrix multiplications. The flow chart is shown in Figure 1.12.

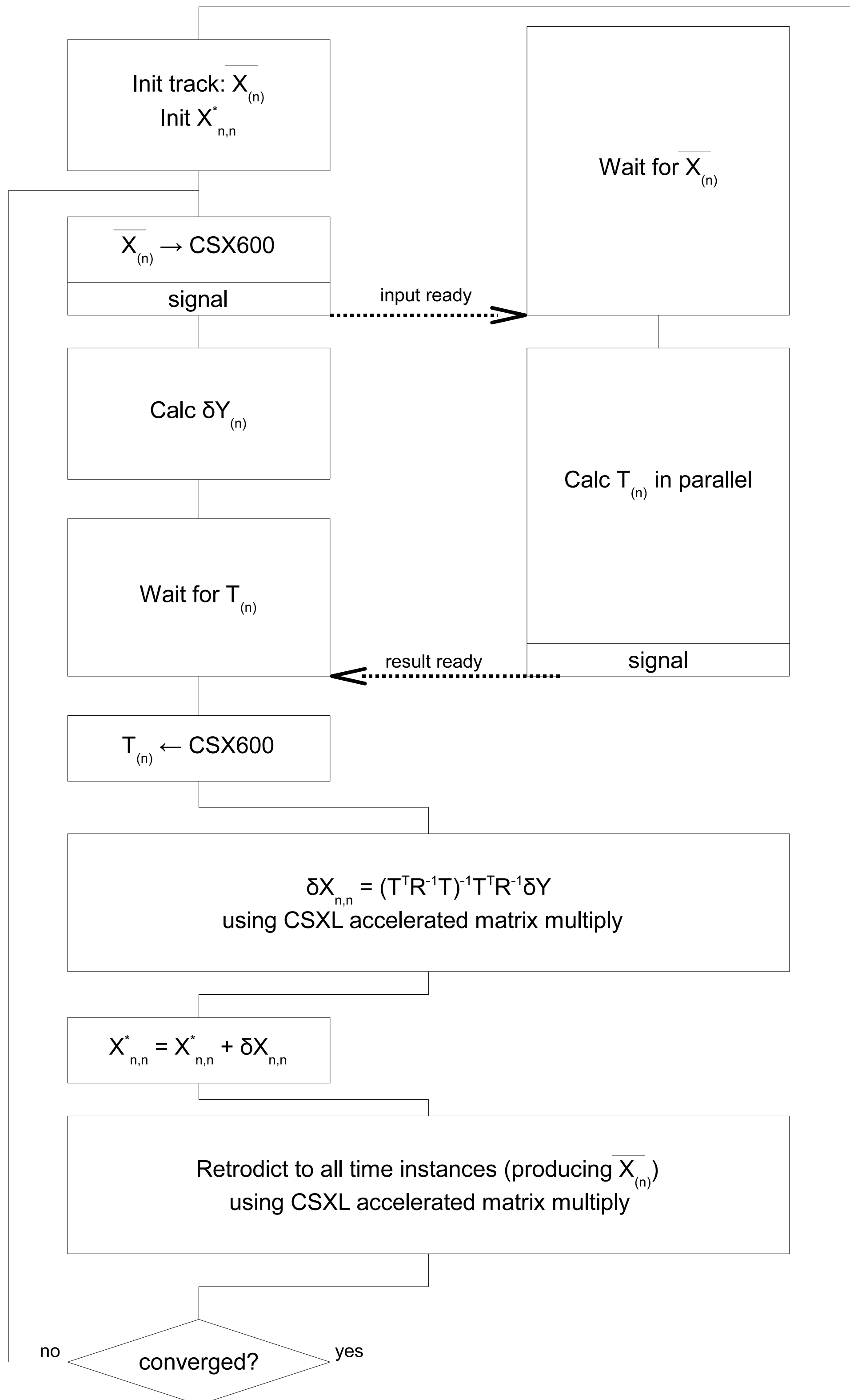

Figure 1.12: Intended ClearSpeed accelerated Gauss Newton implementation flow of control. The sections of code identified as candidates for ClearSpeed acceleration were those involving loops with no data dependency between iterations, and the high dimension double precision matrix multiplications.

The performance prediction of the total observation matrix calculation on ClearSpeed involved counting the instructions and multiplying by the expected execution time of each instruction [15]). The transfer of data to and from the card were also factored in, the expected transfer times discovered experimentally. The total execution time including data transfer for the total observation matrix was 5.27 ms for a single iteration. The C implementation observation matrix and observation perturbation vector calculation took on average almost 20 ms, therefore the ClearSpeed implementation was expected to give a 3 to 4 times increase in performance.

The matrix multiplications involved were found to not be well suited for ClearSpeed acceleration. Testing the speed of CSXL library calls against the host ATLAS libraries showed that all of the matrix multiplications involved were not a fit, as is shown in 1.13. In most of the matrix sizes, fairly dramatic performance degradation was seen. This illustrates the main drawback to co-processor acceleration, that when the problem size or type does not fit the hardware of the co-processor, no significant performance benefit will be seen. In some cases, as in this one, performance degradation can even result. Consider the matrix multiplication: $C = A * B$ where A is an m by k matrix, $B$is a k by n matrix and C is the m by n matrix resulting from the multiplication of the two. ClearSpeed's DGEMM routine performs best when m and n are multiples of 192 and k is a multiple of 288 [11]. The only situation where any of these conditions are satisfied is the first multiply of the state vector perturbation vector (labeled *covMat x TTR-1* in Figure 1.13) where n=1920, however m=k=33 and the performance is still inferior to ATLAS.

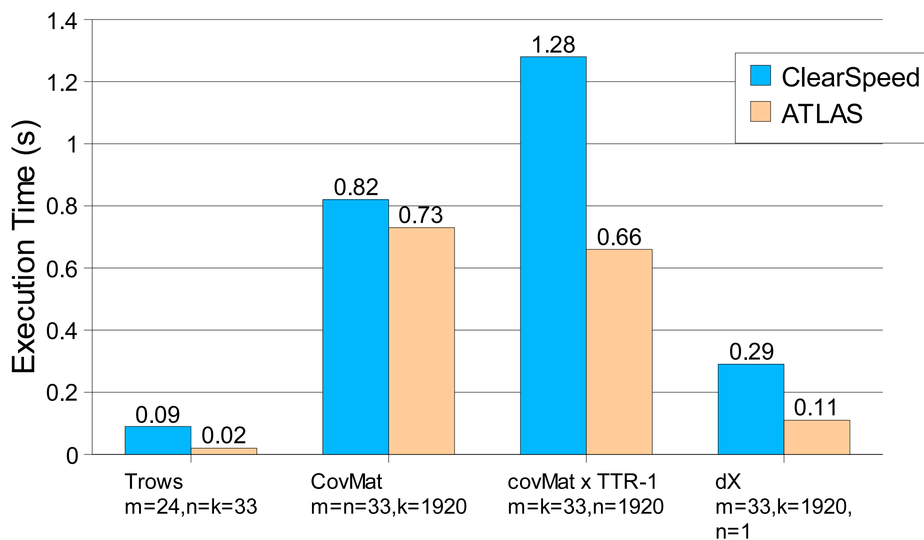

Figure 1.13: CSXL vs ATLAS matrix multiplication execution times. The main matrix multiplications of the algorithm were tested for performance, the graph shows execution times for 500 iterations of each multiplication. Here the symbols m, n and k represent the dimensions of the matrices in the multiplication where an m by k matrix has been multiplied by a k by n matrix, the resulting matrix being m by n. *Trows* represent the multiplication of the sensitivity matrix for a sample $M(X_i)$ (24 by 33), with the transition matrix for that sample Φ_i (33 by 33). *CovMat* represents the inverse covariance matrix calculation where $(T_{(n)}^T R_{(n)}^{-1})$ (33 by 1920) is multiplied by $T_{(n)}$ (1920 by 33). *covMat x TTR-1* is the multiplication of the covariance matrix, $(T_{(n)}^T R_{(n)}^{-1} T_{(n)})^{-1}$ (33 by 33), with $(T_{(n)}^T R_{(n)}^{-1})$ (33 by 1920). The perturbation vector *dX* is calculated by multiplying the 33 by 1920 result of the previous multiplication with the total observation perturbation vector $\delta Y_{(n)}$ (1920 by 1).

Implementation and verification

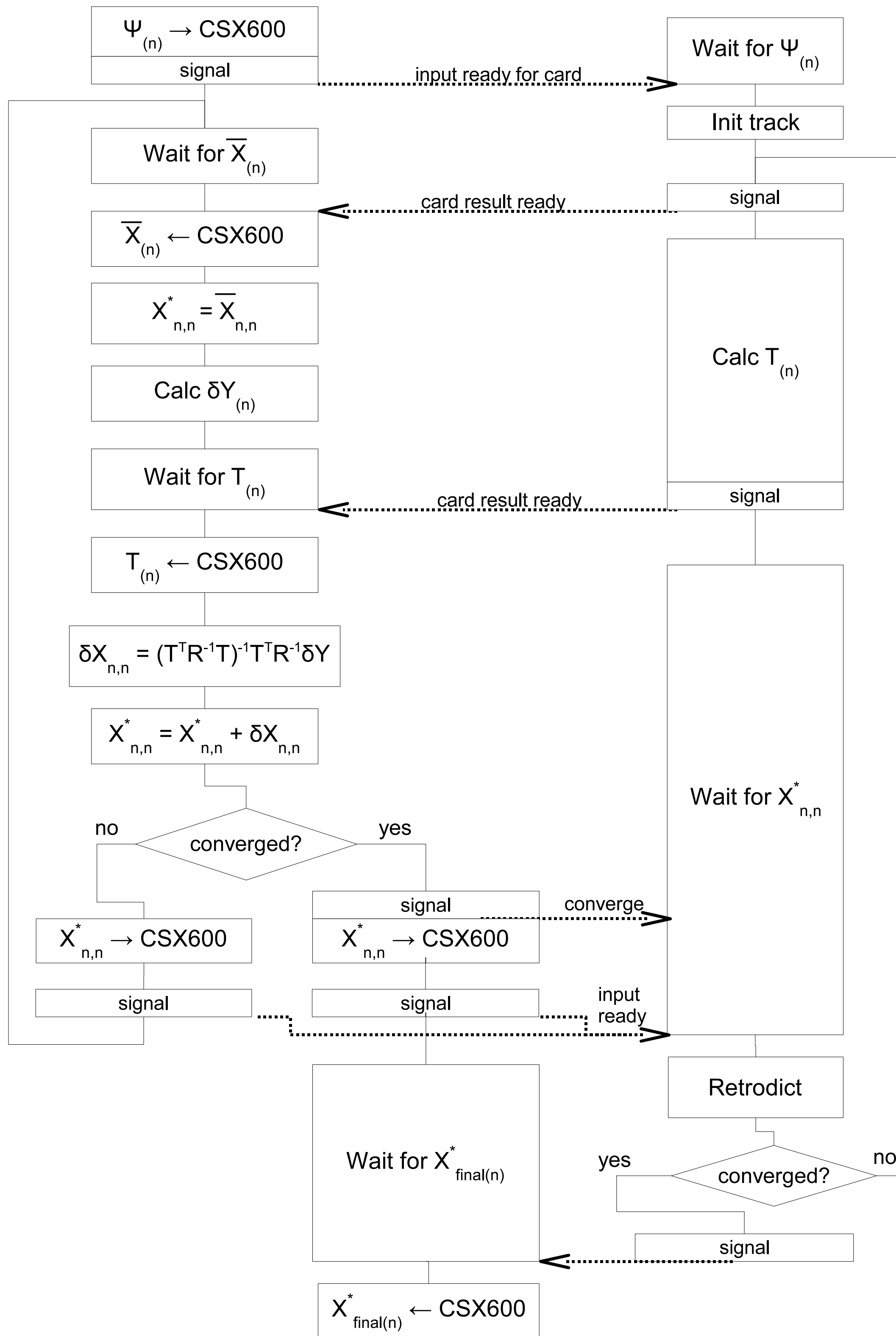

Figure 1.14: ClearSpeed accelerated Gauss Newton flow chart. Computations offloaded to the ClearSpeed card were track initialization, calculation of the total observation matrix and state vector retrodiction. Acceleration was attempted by removing redundant data and unrolling loops.

The implementation phase involved iterative coding and verification using black box testing on a functional unit by functional unit basis. This was done by sending relevant test data to the ClearSpeed card, running the unit in question on the co-processor, then reading the output and testing for accuracy. If an error could not be identified by code examination after a failed unit test, the ClearSpeed debugger was used. This allowed tracing through the flow of control of the co-processor accelerated unit. Simultaneous tracing through the unassisted code allowed comparison of the sequential state of the host and accelerated versions in order to identify the erroneous section of code. The ClearSpeed implementation flowchart is shown in Figure 1.14. The accuracy of the ClearSpeed accelerated algorithm was tested in a similar fashion to the C implementation. The accuracy results verify the correct functionality of the ClearSpeed accelerated algorithm (Figure 1.15), however the results showed that the ClearSpeed accelerated implementation produced estimates an order of magnitude less accurate than the IDL version. This can be attributed to the differences in accuracy of the card side matrix multiplication and the IDL and BLAS library routines. This is supported by examining the average values of the perturbation vector ($\delta X_{(n)}$), which show that in the accelerated version perturbation vector was on average larger than that of the unaccelerated algorithm (Table 1.2).

Table 1.2: Comparative stopping rule statistics of C implementation and ClearSpeed assisted algorithm. The larger average perturbation vector of the accelerated implementation suggests that the ATLAS BLAS library implementation of matrix multiplication is more accurate than the custom written card side calculation.

	Iterations	average $\delta X_{(n)}$	Doppler residuals	bearings residuals
C	4.49162	5.46657	7.18154	0.00658
CSX	4.52189	5.65714	7.19699	0.00658

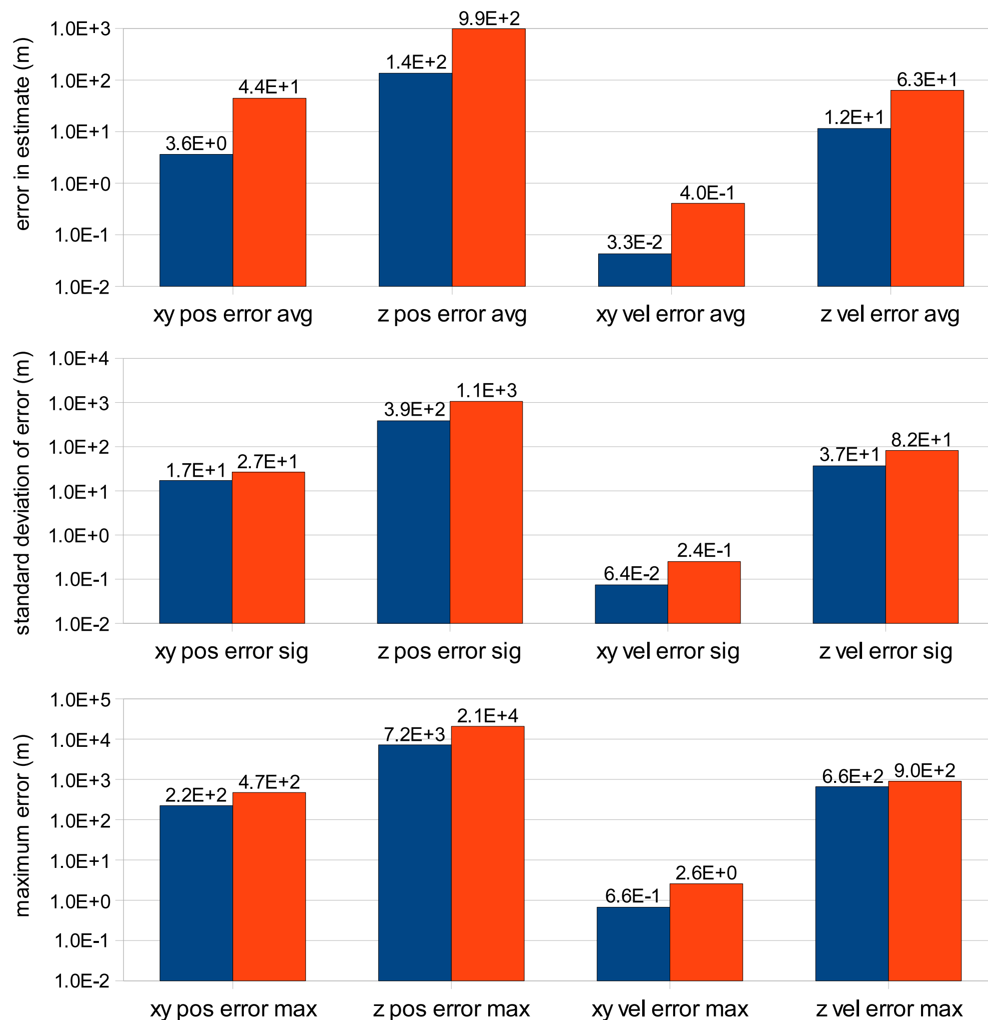

Figure 1.15: ClearSpeed accelerated accuracy results. The ClearSpeed results are an order of magnitude less accurate than those produced by the original IDL implementation, however still sufficiently to meet the stopping conditions of the algorithm as defined by Morrison [3]

ClearSpeed accelerated implementation profiling

The ClearSpeed accelerated algorithm was profiled using the ClearSpeed debugger. The output was viewed with the ClearSpeed visual profiler, and processed using a spreadsheet program to allow its performance to be compared with the host version. The results are shown in Figures 1.16 and 1.17, which show the profile of the host and ClearSpeed interactions and processing. The graphs are colour coded to show simultaneous host co-processor execution.

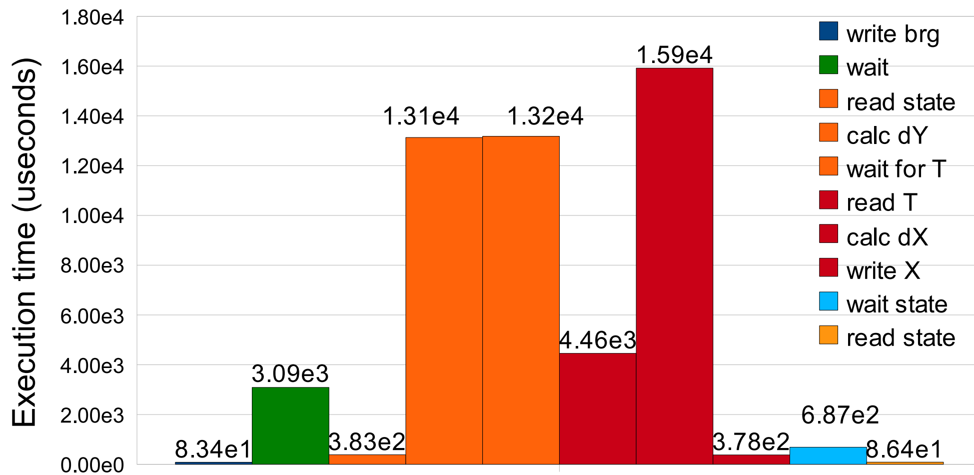

Figure 1.16: ClearSpeed accelerated host processing profile. Note that the first block (navy) is the *write brg* block that writes the bearing observations to the ClearSpeed card. The *write X* block writes the state vector (10th degree polynomial in 3 dimensions) to the card. The final *read state* block (yellow) reads the position and velocity retrodicted to all time instances in 3 dimensions back from the card.

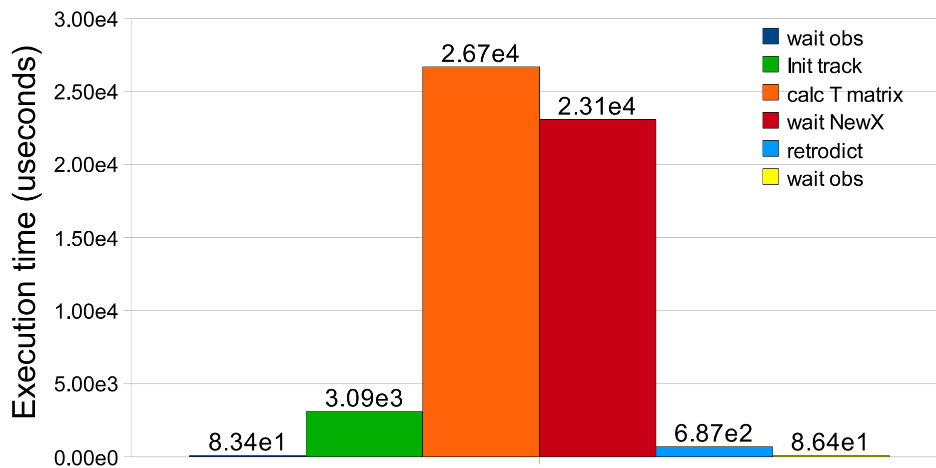

Figure 1.17: ClearSpeed accelerated card side processing profile. Note that the first and last block (navy and yellow respectfully) are the *wait obs* blocks in which the card waits for the bearings observations from the host. The first *wait obs* block happens while the bearings are being written to the card, and the last one occurs while the position and velocity estimate is being read back from the card.

A graphical representation of the performance increase of the various sections of code is shown in Figure 1.18.

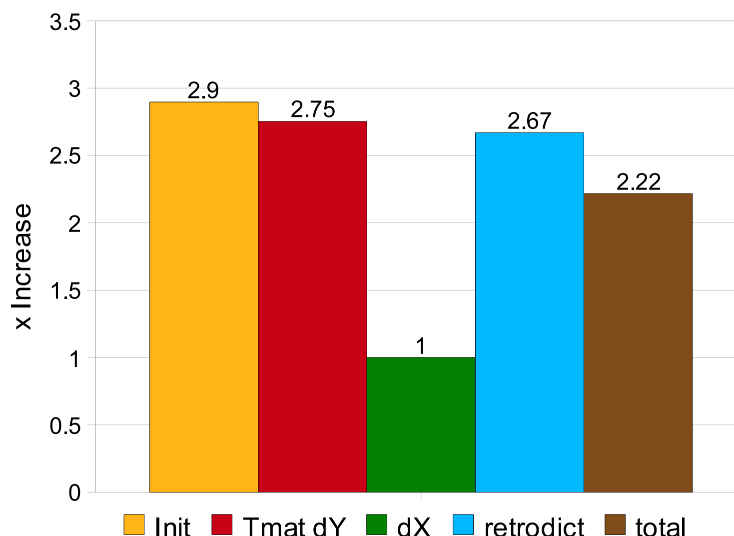

Figure 1.18: ClearSpeed accelerated performance increase graph

Conclusions

The accuracy results verify the correct functionality of the ClearSpeed accelerated algorithm, however the results showed that the ClearSpeed accelerated implementation produced estimates an order of magnitude less accurate than the IDL version. The final implementation resulted in about a 2.22 times speedup on average.

1.4.7 Conclusions

In chapter 7 conclusions are drawn and the results of the co-processor offloading implementation are discussed. Suggestions are also made concerning future work that can be done in the acceleration of the Gauss-Newton tracking algorithm.

Limitations of implementation

The implementation was capable of estimating target position and velocity in two dimensions. However target altitude estimations were not particularly accurate under the simulation conditions used. This was due to the limited number of receivers used in the simulated radar system. If more receivers were added to the PCL network, the input data set would rapidly multiply to an impractical size considering the limited available poly memory.

The accelerated Gauss-Newton implementation was tested for accuracy against the IDL version only. Therefore its functionality in the field is contingent on the real-world applicability of the pre-existing IDL code, which is yet to be tested. The IDL radar data generator used is fairly simplistic, therefore neither implementation has been tested with more realistic data.

Comments on programming with ClearSpeed

The ClearSpeed programming model has a two to three week learning curve, as it is fairly straight forward. Users experienced in parallel programming will find ClearSpeed programming even easier to adjust to. Due to the hefty price

tag of the card, significant performance gains are required to make porting applications feasible. These performance gains will only be seen if the problem is either data parallel with high computation to data transfer ratio, or if linear algebra problems of a specific size are being solved. The amount of poly memory present is a major constraint, as is the bandwidth between mono and poly memory. These two constraints often result in performance penalties as the program is forced to continually swap data between mono and poly memory if the data set is too large. Single or even half precision should be used wherever possible.

Results of acceleration

Direct acceleration of the matrix operations via the provided ClearSpeed accelerated library functions was not feasible. The sizes of the matrices used were not conducive to ClearSpeed acceleration, even after tweaking the host assist environment variable (refer to 2.3). Further investigation concluded that a moderate speedup could be attained through parallelization of the total observation matrix calculation. This acceleration was possible partly due to the sparse and data redundant nature of the input matrices. This allowed redundant data and calculations to be eliminated in the matrix multiplication. The CSXL library implements standard BLAS algorithms and therefore does not do any matrix value analysis and all matrices are assumed to be dense. The reasoning behind this is that the processing penalty in analyzing matrix values at runtime would outweigh the performance benefit that would result. Optimization at the programming stage can result in performance benefits but requires in depth knowledge and understanding of the algorithm and its data structures. The trade off is always between performance benefit and man hours spent tweaking the code. The final implementation resulted in about a 2.22 times speedup on average. The $T_{(n)}$ matrix and $\delta Y_{(n)}$ vector calculation (executed in tandem on host and co-processor) had an approximate 2.75 times speedup on average. The design stage predicted an execution time of 5.14e3 microseconds for a single iteration of the $T_{(n)}$ and $\delta Y_{(n)}$ calculation, whereas the implementation averaged around 5.52e3 microseconds, a difference arising from the assumptions made in the performance prediction (see section 6.1.2).

Future work

Further work could be done in making the implementation more general, allowing for more receivers or different aircraft and radar parameters. Accounting for additional receivers would be particularly difficult as the data structures involved would increase in size significantly, however the second CSX600 processor on the board could perhaps be used to allow for this. Additionally it could be worth considering the implementation of a card-side matrix multiplication application fine tuned specifically for the matrix dimensions involved in the Gauss-Newton algorithm.

Given the expensive nature of the ClearSpeed cards, and the ultimate goal of reducing costs through PCL tracking, it would be worth considering cheaper alternative options to accelerate the Gauss-Newton filtering algorithms. This consideration is further motivated by the fairly moderate speedup achieved by the implementation, and the relative inaccuracy of the tracking estimates produced. These might include options such as GPGPU's and FPGA technologies, although the requirement for double precision might be restricting if FPGA's are to be used. The ClearSpeed accelerated code developed in this project might be used as a starting point for such an investigation or implementation.

For the Gauss-Newton tracking algorithm to ultimately be useful in the field, it should be able to track multiple targets simultaneously. While this appears to be an inherently parallel problem, the multiple target filtering algorithm still requires further work. Problems that are still to be solved centre on track identification (how to identify which data belongs to which target).

Chapter 2

Description of hardware

The ClearSpeed co-processor technology used is described in this section. Acceleration via co-processor offloading is introduced, the ClearSpeed Advance board is described and an overview of the ClearSpeed software environment is provided.

2.1 Co-processor assisted acceleration

The traditional method of increasing computing performance by increasing clock speed and transistor density of a von Neumann architecture CPU is now facing a brick wall limit in terms of the amount of energy that a single microprocessor die can dissipate [1]. Therefore innovations in computer architecture and the replacement of fast, monolithic, single processor dies with multiple slower processors operating in parallel must substitute for the traditional method [1].

One approach to increasing compute performance is the use of co-processor accelerator boards. Such boards can offer dramatic power and space benefits over other forms of HPC while offering impressive performance increases over conventional serial computing architectures, if the algorithm in question is well suited to the co-processor hardware architecture.

2.2 The ClearSpeed Advance X620 board

The co-processor used for this project was the ClearSpeed Advance X620 accelerator board, which fits into the PCI-X slot of a conventional computing platform. The platform used had a 133MHz PCI-X 2.0 slot. It is claimed that the Advance board range uses the fastest and most power efficient double-precision 64-bit floating point processors in the world. The X620 consists of two CSX600 embedded parallel processors communicating via the ClearConnect busbridge ports, 1GB of local DDR2-400 SD-RAM and an FPGA which implements the host interface [13]. A block diagram of the Advance X620 architecture is shown in Figure 2.1.

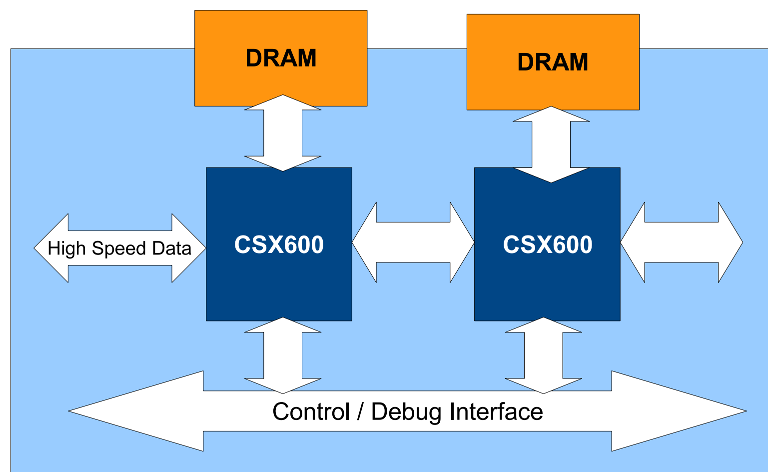

Figure 2.1: ClearSpeed Advance X620 architecture block diagram. The X620 consists of two CSX600 embedded parallel processors communicating via the ClearConnect busbridge ports, 1GB of local DDR2-400 SD-RAM and an FPGA which implements the host interface [13]

The FPGA is only reprogrammed when upgrading the ClearSpeed firmware for updates or new functionality.

2.2.1 The CSX600 parallel processor

The CSX600 parallel processor is shown in Figure 2.2. The CSX600 is a fully integrated system on a chip (SoC). The major components include:

- A Multi-Threaded Array Processor (MTAP)[1].
- A DMA memory controller with a 64-bit interface to the on board DDR2 memory. With 64-bit addressing the CSX600 supports multi-gigabyte off chip SDRAM. The Advance board has 512MB of SDRAM per processor.
- 128kB scratchpad on-chip eSRAM memory.
- The Host interface and Debug Port (HDP).
- An Interrupt and Semaphore Unit (ISU).
- The proprietary ClearConnect NoC that provides on chip and inter-chip (via the ClearConnect BusBridge Ports (CCBR)) communication.

[1]MTAP is a type of processor which allows multiple, disparate instructions to be executed simultaneously. Hence the term Multi-Threaded. For example a single processing element may contain a floating point adder, a floating point multiplier, an integer ALU and an integer MAC. Theoretically in an MTAP each of these disparate execution units within the processing element could execute a separate thread of execution simultaneously, as long as there is data independence between threads.

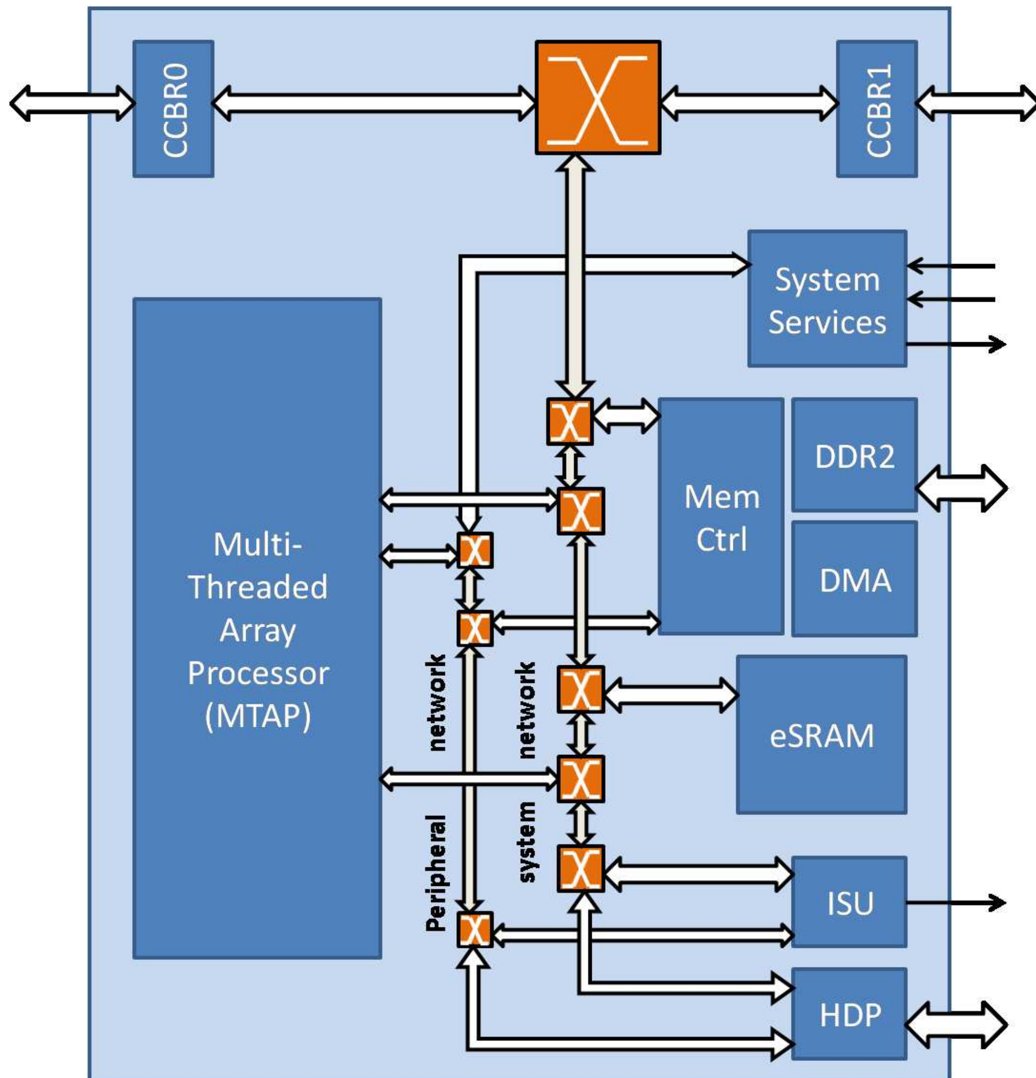

Figure 2.2: ClearSpeed CSX600 processor architecture block diagram[7]

The CSX600 provides 25GFLOPS of sustained single or double precision performance while dissipating an average of 10W [7], which is an astonishing performance per W of 2.5GFLOPS/W.

The MTAP which constitutes the processing core of the CSX600 is shown in Figure 2.3. The MTAP consists of a single instruction stream array processor. When each instruction is decoded it is executed by either the mono processor (single instruction single data execution, as per the traditional von Neumann architecture) or the poly execution unit (single instruction multiple data). The poly execution unit consists of an array of 96 processing elements (PEs). Each processing element consists of the following: [7]

- 32/64-bit floating point multiplier
- 32/64-bit floating point adder
- Divide / square root unit
- Integer ALU and 16-bit integer MAC

- 128-bit register file
- 6kB of SRAM

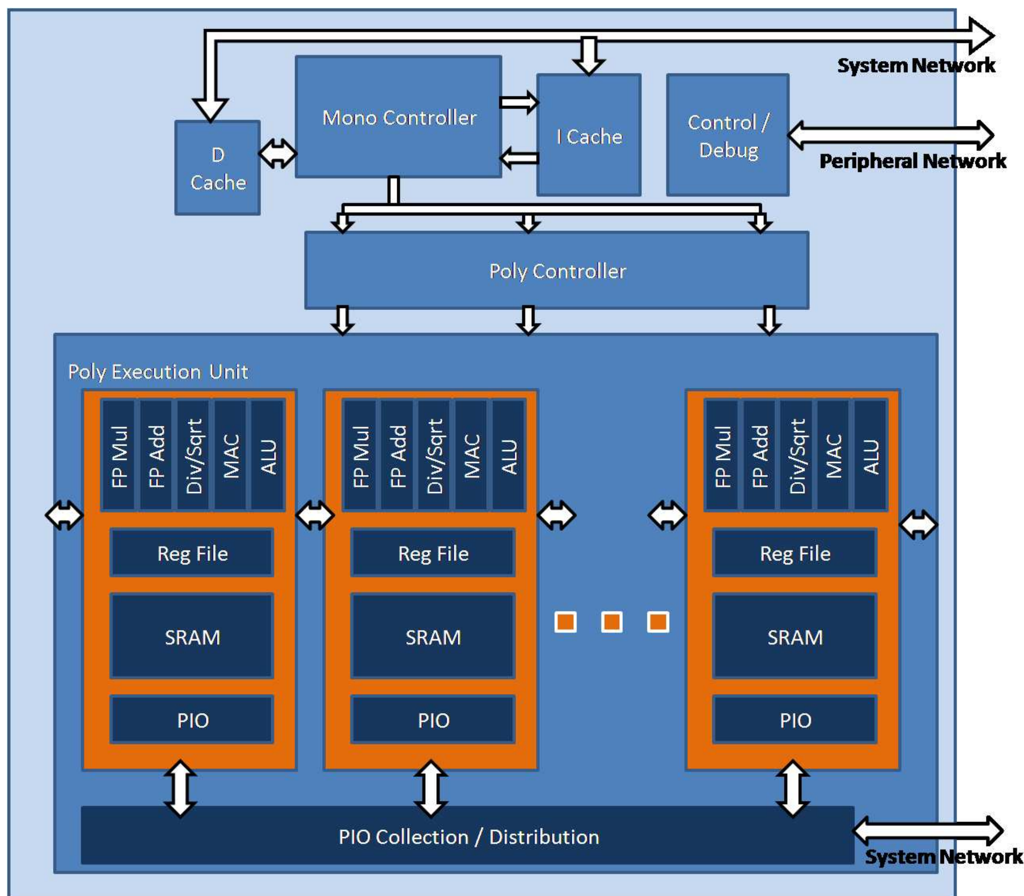

Figure 2.3: The CSX600 Multi-Threaded Array Processor (MTAP)[7]

2.3 The ClearSpeed software environment

The ClearSpeed software environment consists of the runtime libraries for interfacing with the board (including device drivers), the CSXL and CSDFT application acceleration libraries, and the Software Development Kit (SDK). The SDK consists of a C/C++ compiler with parallel extensions, standard C libraries, instruction set simulator, industry standard debugger, visual profiler, vector math library, random number generator and tools for loading and running code onto the CSX600. The relationship between the different parts of the software environment is shown in Figure 2.4.

Application acceleration with the ClearSpeed Advance X620 is achieved via two methods, parallelization and calling accelerated versions of various linear algebra routines (or FFT routines).

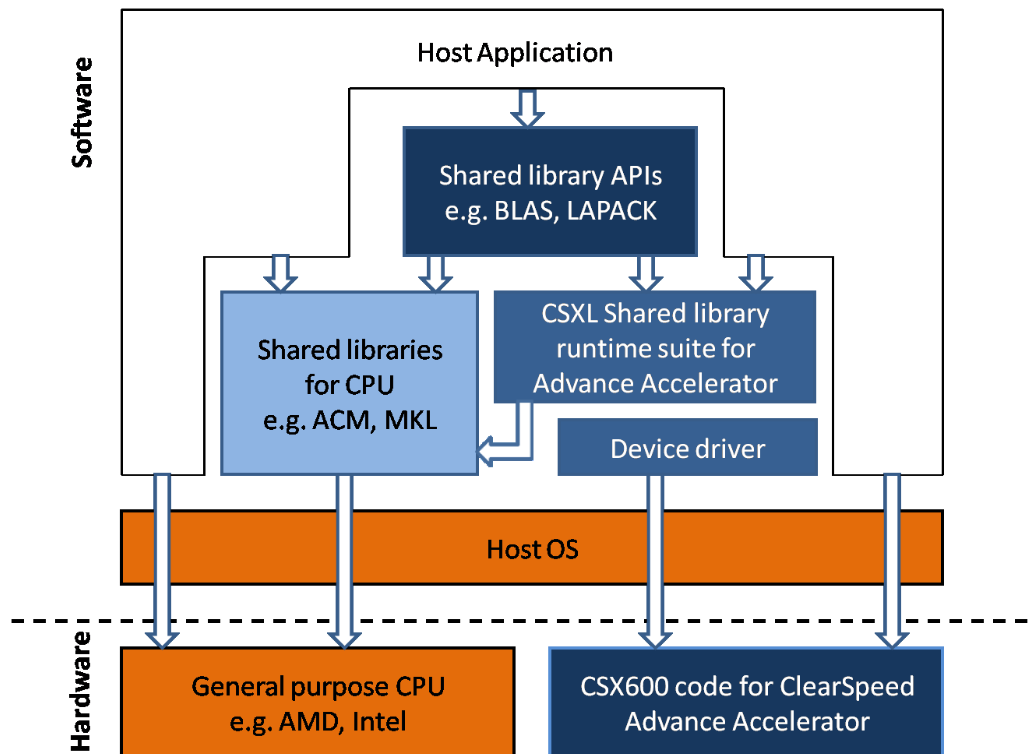

Figure 2.4: ClearSpeed software architecture[13]

The CSX600 processor runtime package provides for the loading and running of standalone ClearSpeed programs, as well as a programming interface and set of libraries that allow host programs to control and communicate with the CSX600 processor.The tool-set for loading and running standalone ClearSpeed programs will not be treated here but is described in detail in the ClearSpeed runtime user guide [6].

Host programs written in C interact with the ClearSpeed hardware in two ways, either via the provided ClearSpeed accelerated library functions or via user programs written in the ClearSpeed Cn language. Cn is based on ANSI C, but adapted and extended for parallel computing with ClearSpeed. The ClearSpeed provides two library interfaces for acceleration of specific linear algebra and DSP functions, the CSXL and CSDFT libraries. The CSXL library supports acceleration of a subset of level 3 BLAS and LAPACK linear algebra routines, and the CSDFT library provides acceleration of various FFT related functions. Since the Gauss-Newton algorithm examined in this dissertation does not make use of any FFT related functions it will not be discussed here. Only the CSXL library is of any relevance and is treated in section 2.3.1.

Host - CSX processor co-operation is implemented through ClearSpeed's CSAPI interface as part of the runtime package driver library. which a host program uses to load, run and communicate with code on the ClearSpeed hardware. The use of CSAPI is described in section 2.3.2.

2.3.1 Acceleration with the CSXL library

ClearSpeed's CSXL library provides accelerated versions of various mathematical functions for use with the ClearSpeed Advance co-processor. There are two usage models for calling CSXL library routines, either from a host program or directly from Cn code executing on the CSX600 processor.

Calling CSXL functions from the host

The routines that are supported are a subset of the BLAS (Basic Linear Algebra Routines) and LAPACK (Linear Algebra PACKage) libraries. These libraries are used to solve various common linear algebra problems. The functions supported by CSXL are listed below [14].

Supported BLAS functions:

- DGEMM: double precision general matrix multiplication.
- ZGEMM: double precision general matrix multiplication with complex data.
- ZGEMM3M: alternate implementation of ZGEMM that uses three real multiplications instead of ZGEMM's five. May run up to 33% faster than ZGEMM under certain circumstances.
- DTRSM: solves a triangular system of equations with double precision data.

Supported LAPACK functions:

- DGETRF: computes an LU factorization of a matrix using partial pivoting with row interchanges.
- DGETRS: solves a system of linear equations using the LU factorization performed by DGETRF.
- DGESV: computes the solution to a real system of linear equations (combines DGETRF and DGETRS).
- DPOTRF: computes the Cholesky factorization of a real symmetric positive definite A.
- DPOTRS: solves a system of linear equations with a symmetric positive definite matrix using the Cholesky factorization.
- DPOSV: computes the solution to a real system of linear equations (combines DPOTRF and DPOTRS).
- DGEQRF: computes a QR factorization of a real matrix.
- DORGQR: generates a real matrix with orthonormal columns which is defined as the first N columns of a product of K elementary reflectors of order M as returned by DGEQRF.
- DORMQR: Overwrites the general real M-by-N matrix C with the following where Q is a real orthogonal matrix, defined as the product of k elementary reflectors as returned by DGEQRF:

SIDE = 'L' SIDE = 'R'

TRANS = 'N' : Q * C C * Q

TRANS = 'T' : Q**T * C C * Q**T

There are two ways to use the CSXL library, either from a host or from a CSX600 application. The host side library requires that an implementation of the BLAS and LAPACK libraries is installed on the machine. When a CSXL library supported function is called by the host program, the runtime environment intercepts the function call and determines how to split the processing between the host and co-processor using heuristics. The environment variable

CS_BLAS_HOST_ASSIST_PERCENTAGE is used as a guide when determining how much processing to offload, however the final decision lies with the runtime environment. The optimal value of CS_BLAS_HOST_ASSIST_PERCENTAGE for a particular system is determined by a script provided by ClearSpeed.

Only certain matrix dimension sizes will result in performance benefits when using the CSXL library functions. Figure 2.5 shows a host application calling a CSXL supported BLAS function, both with just a host BLAS library function installed, and with a ClearSpeed assisted CSXL library function.

Various host BLAS library implementations have been tested for compatibility with CSXL, including the following [14]:

- ACML
- MKL
- ATLAS

The host BLAS library used here was ATLAS (Automatically Tuned Linear Algebra Subroutines). ATLAS is a free BLAS implementation that is optimized for the host hardware at build time.

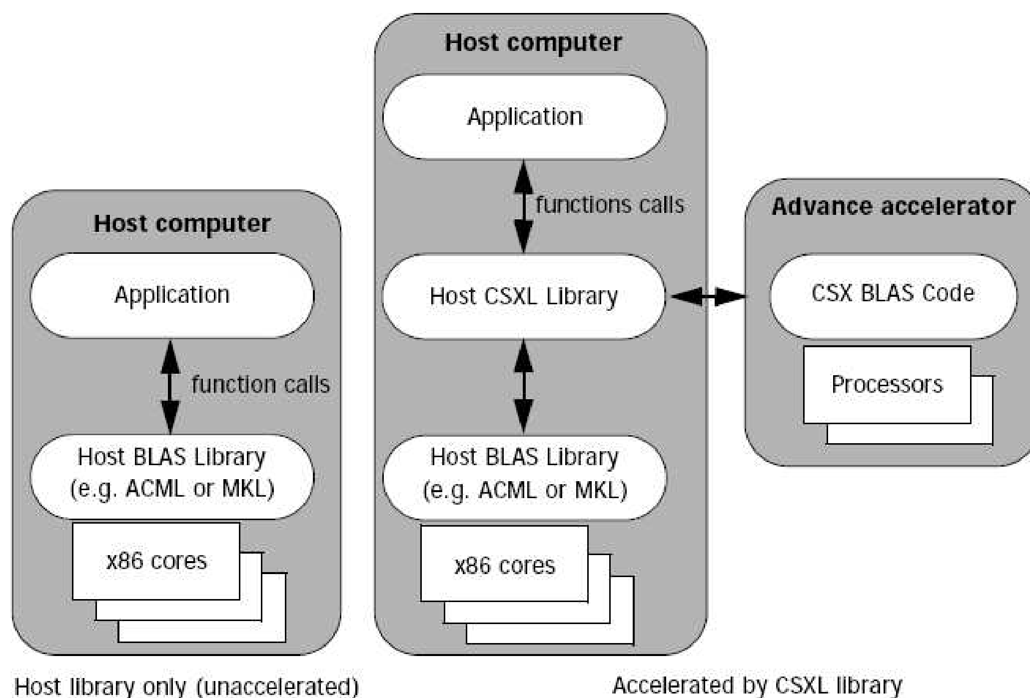

Figure 2.5: Host application calling BLAS functions. When a CSXL library supported function is called by the host program, the runtime environment intercepts the function call and determines how to split the processing between the host and co-processor using heuristics[14]

Calling CSXL functions from Cn code

The only card-side CSXL function available is blocked double precision general matrix multiplication. Only matrices whose number of rows and columns are multiples of 96 can be operated upon. This restriction rules card-side CSXL calls out for the Gauss-Newton implementation under study here.

2.3.2 Acceleration through parallelization

Other than calling the provided CSXL (or CSDFT) library functions, acceleration occurs through parallelization of code. Users may write their own custom ClearSpeed applications. A ClearSpeed accelerated program consists of two components, the host program and the code running on the ClearSpeed hardware (the CSX program) [6]. The ClearSpeed code is written in the Cn programming language. A time-line for typical CSAPI interactions with a host programming controlling code running on the ClearSpeed hardware is shown in Figure 2.6.

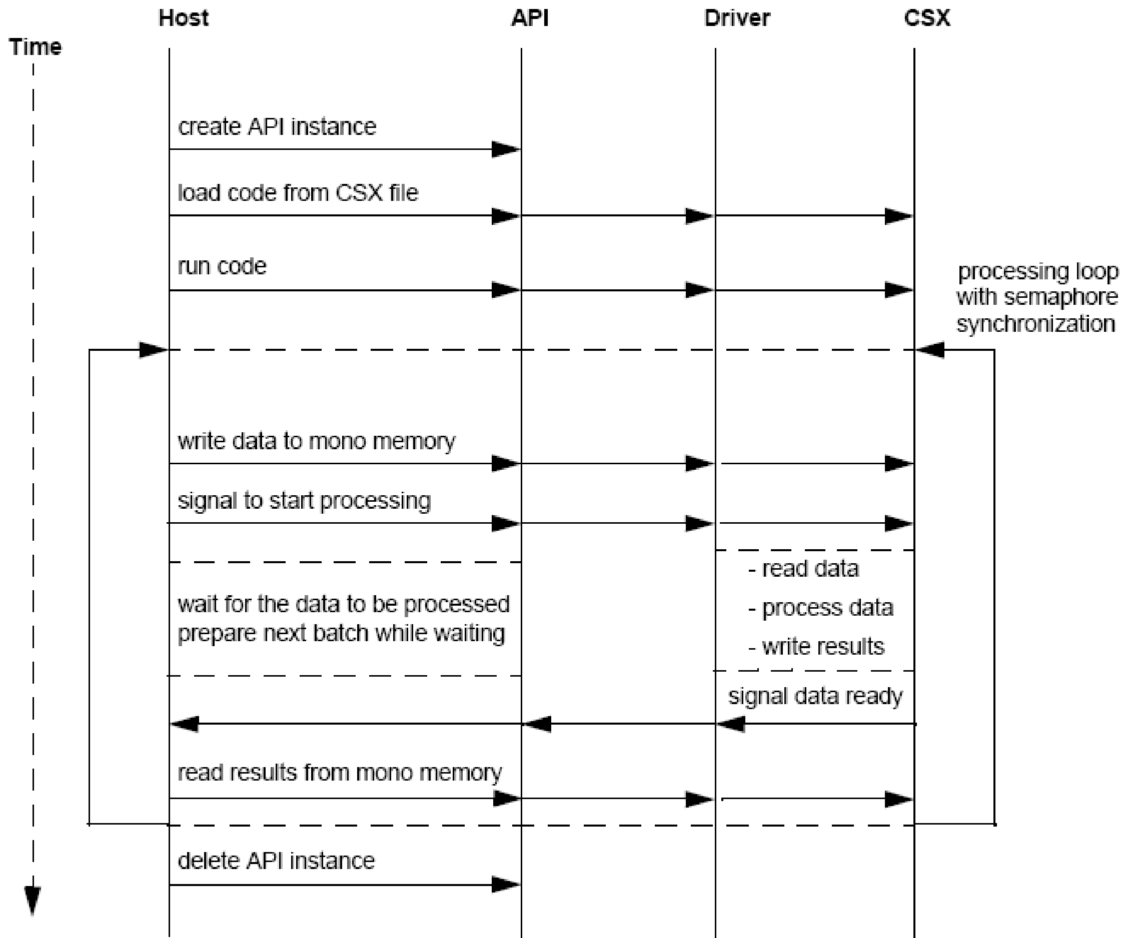

Figure 2.6: CSAPI driver library interaction time-line[6]

The figure shows the basic procedure involved in offloading computation with ClearSpeed.

- Create an instance of the API, to be used with one Advance board.
- Load Cn code onto a specified CSX600 processor on the Advance board.
- Launch the Cn application.
- Begin the processing loop:
  - Write the Cn program's input data onto the Advance SDRAM.

  - Signal a shared semaphore to inform it that the input data is ready and processing may begin.
  - Prepare the next batch of data then wait for the Cn program output to be ready. The host may of-course perform its own independent processing while waiting for the Cn program to complete.
  - On receipt of a signal from the Cn program, read the output data from the co-processor.
  - If there is more data to process, continue looping.

- When the processing loop completes the program is unloaded which releases the resources of the board so that it can be used by another Cn program.

- When the board is no longer required the API instance associated with it is deleted, unloading the CSAPI library.

This is a simplified interaction time-line, with unnecessary complexities removed. However there are several powerful programming concepts that facilitate far better performance benefit than is achievable through the above simple programming model, including semaphores, asynchronous data transfer and double buffering.

The CSAPI library provides access to two types of semaphores, Thread Sequence Controller (TSC) and Global Semaphore Unit (GSU) semaphores. TSC semaphores are used to synchronize execution between Cn code and a host program, or between CSX600 threads. GSU semaphores synchronize processors on the same card, the host DMA engine and the host application. GSU semaphores can also carry data. The Gauss-Newton implementation ran on only one CSX600 processor and did not make use of GSU semaphores. The properties of the two different semaphores types are shown in Table 2.1.

Table 2.1: Semaphore properties[6]

Event	TSC	GSU
Total number on each processor	128	16
Signal overflow limit	255	65536 (32 with data)
Can carry data	No	Yes
Can be operated on by other processors	No	Yes

The CSX600 parallel programming paradigm utilizes a single instruction stream with multiple data. The CSX600 architecture uses an array of processing elements that make up the poly execution unit. Programming the CSX600 processor for parallel execution is based on the concept of the poly data type specifier. A poly specifier results in the data being stored in poly memory, and functions that act on poly data will be executed by the poly array of processing elements. What is most pertinent to the application programmer about the CSX600 design is that there is a mono and poly execution unit, and that each instruction in the single instruction stream will be executed by one of the two (but not both) as appropriate [12]. Mixing of poly and mono data types can result in undesired behaviour if care is not taken.

A poly data type has many instances, one on each poly execution unit (PE). When poly code executes, each PE performs the same function from the instruction stream on its own data stored in memory local to that PE (poly memory). The SDK includes most standard C libraries, with mono and poly variants.

Conclusions

In order to be candidates for parallelization, applications need to be data parallel, have a high computation to data movement ratio and small working data sets [8]. Keeping within the memory requirements of the CSX600 requires some care, and performance is vastly improved if it is possible to overlap computation and data transfer [10]. This goes for communication between the host and co-processor as well as for data transfer between the various tiers of on-board memory.

An application is also a candidate for acceleration if a high percentage of application execution time is taken up by linear algebra routines supported by ClearSpeed's CSXL library (or the CSDFT library which performs various FFT functions). When the host application calls a library routine, the runtime environment uses heuristics to determine whether to run that routine on the host or the Advance board [14]. Performance benefits are only significant when the problem size is a good fit for the ClearSpeed hardware. An example of this is the ClearSpeed accelerated DGEMM library function. Consider the matrix multiplication: $C = A * B$ where A is an m by k matrix, $B$is a k by n matrix and C is the m by n matrix resulting from the multiplication of the two. ClearSpeed's DGEMM routine performs best when m and n are multiples of 192 and k is a multiple of 288 [14].

Chapter 3

The Gauss-Newton tracking algorithm

This chapter describes the key filter engineering concepts and mathematical theorems involved in the development of the Gauss-Newton tracking algorithm. The chapter begins by defining some of the notation used in the mathematical equations herein, then goes on to introduce the reader to Gauss-Newton filtering by giving an overview of what it entails. The core concepts involved in Gauss-Newton filtering are then addressed in some detail. These are the observation equations, the observation or T matrix calculation and the minimum variance rule. Once this background material has been presented the iterative Gauss-Newton filtering method that was used for this project is outlined.

3.1 Definitions

In describing the Gauss-Newton filtering and tracking process the following conventions are used:

- Long column vectors are represented as the transpose of a row vector to save space. Thus $Y = (\; y_1 \quad , \quad y_2 \;)^T$ represents the column vector $\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$.
- t_n denotes "right now", or the most recent observation instant.
- Y_n is a vector that is valid for time instant t_n, the most recent observation instant.
- $Y_{(n)}$ is a concatenation of the vectors valid from time t_L up to the most recent observation instant t_n.
- t_L is the initial observation instant used for the filtering cycle.
- Y_{n_k} is the observation vector for the kth receiver at time t_n.
- $\dot{x}$ is the 1st time derivative of the variable x, $\ddot{x}$ the 2nd time derivative and so forth, as is the convention in calculus.
- $D^m x$ denotes the mth time derivative of the variable x. The operator D is thus the differential operator.

3.2 The observation equations

Consider an aircraft passing through the target area of a PCL radar system consisting of 8 receivers; Let the vector of observations made at time t_n be Y_n be described by equation 3.1 [3].

$$Y_n = (\ f_d \ , \ \cos(\psi) \ , \ \sin(\psi)\ )^T \tag{3.1}$$

Each parameter of Y_n is a vector made up of 8 observation scalars. This is because there are 8 receivers and therefore 8 sets of observations at any particular instant. f_d is a set of Doppler shift observations, $\cos(\psi)$ and $\sin(\psi)$ are the cosine and sine of the bearing angle observations respectively[1]. Note that this particular observation vector is used for convenience of discussion. In the general case Y_n could contain any number of observation variables.

In a real world situation the observations made by the receiver network will be noisy. The vector of errors in the observation vector Y_n is represented by N_n in equation 3.2 where each of v_1, v_2 and v_3 is a vector of 8 scalars.

$$N_n = (\ v_1 \ , \ v_2 \ , \ v_3\ )^T \tag{3.2}$$

The complexity of the T matrix calculation depends on the linearity of the observation scheme and the filter model used to estimate the state of the observed system [3]. Here the observation scheme is nonlinear, as shown by equation 3.4. The filter model is assumed to be linear. The only linear filter model of practical interest is the polynomial [3]. The Gauss-Newton implementation models the state of the tracked aircraft as a degree ten polynomial, and thus the state vector consists of the 0th to 10th derivatives of the estimated x, y and z co-ordinates of the aircraft. The state vector for the aircraft at time t_n is described by equation 3.3.

$$X_n = (\ x,\ \dot{x},\ \ddot{x},\ \ldots\ D^{10}x,\quad y,\ \dot{y},\ \ddot{y},\ \ldots\ D^{10}y,\quad z,\ \dot{z},\ \ddot{z},\ \ldots\ D^{10}z\ )^T \tag{3.3}$$

The set of observation equations for the kth receiver which is located at $(\ x_k \ , \ y_k \ , \ z_k\ )^T$ can then be developed. If the transmitter is located at $(\ x_0 \ , \ y_0 \ , \ z_0\ )^T$ and is transmitting at wavelength λ, the observation equation for the kth receiver is shown by equation 3.4[3].

$$Y_{n_k} = \begin{pmatrix} f_{d_k} \\ \cos(\psi)_k \\ \sin(\psi)_k \end{pmatrix} = \begin{pmatrix} -2\pi/\lambda \left[\frac{(x-x_0)\dot{x}+(y-y_0)\dot{y}+(z-z_0)\dot{z}}{\left((x-x_0)^2+(y-y_0)^2+(z-z_0)^2\right)^{1/2}} + \frac{(x-x_k)\dot{x}+(y-y_k)\dot{y}+(z-z_k)\dot{z}}{\left((x-x_k)^2+(y-y_k)^2+(z-z_k)^2\right)^{1/2}} \right] \\ \frac{(x-x_k)}{((x-x_k)^2+(y-y_k)^2)^{\frac{1}{2}}} \\ \frac{(y-y_k)}{((x-x_k)^2+(y-y_k)^2)^{\frac{1}{2}}} \end{pmatrix} + \begin{pmatrix} v_{1_k} \\ v_{2_k} \\ v_{3_k} \end{pmatrix} \tag{3.4}$$

Equation 3.4 can be rewritten as a set of functions of the state vector, as in equation 3.5. In writing $g_{1_k}$the subscript k simply implies that we are dealing with the kth receiver and therefore use its co-ordinates for the constants x_k, y_k and z_k.

$$Y_{n_k} = \begin{pmatrix} f_{d_k} \\ \cos(\psi)_k \\ \sin(\psi)_k \end{pmatrix} = \begin{pmatrix} g_{1_k}(X_n) \\ g_{2_k}(X_n) \\ g_{3_k}(X_n) \end{pmatrix} + \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \tag{3.5}$$

To get the full equation for each of the three observation variables, the eight equations for each receiver are concatenated. This is illustrated by equations 3.6, 3.7 and 3.8.

$$[g_1(X_n)]_k = g_{1_k}|_{X_n} \qquad k = 1\ldots 8 \tag{3.6}$$

[1]The bearing angle ψ is split into its cosine and sine to avoid the discontinuity at $\pm\pi$

$$[g_2(X_n)]_k = g_{2_k}|_{X_n} \qquad k = 1\ldots8 \tag{3.7}$$

$$[g_3(X_n)]_k = g_{3_k}|_{X_n} \qquad k = 1\ldots8 \tag{3.8}$$

In words, equation 3.6 says that the kth element of the column vector of functions g_1 is given by the observation equation for the kth receiver of the network evaluated using the elements of X_n.

The observation equation for the radar system is then given by equation 3.9.

$$Y_n = \begin{pmatrix} g_1(X_n) \\ g_2(X_n) \\ g_3(X_n) \end{pmatrix} + \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = G(X_n) + N_n \tag{3.9}$$

3.3 T matrix calculation

The most challenging and computationally expensive part of the Gauss-Newton algorithm is the calculation of the total observation matrix or T matrix, which is central to the estimation procedure. The T matrix calculation uses the observation sensitivity matrix for a particular time instant, which can then be retrodicted to all time instances using the polynomial transition matrix $\Phi(\zeta)$[3]. In what follows a proof that the polynomial transition works on the perturbation vector is offered, (and the perturbation vector itself is defined), then the observation sensitivity matrix and then the T matrix are developed using the initial proof.

3.3.1 The polynomial transition matrix and the perturbation vector

A nominal state vector for the target $\bar{X}_n$, which is close to X_n in the sense of δX_n, is assumed and is shown by formula 3.10 [3].

$$\delta X_n = X_n - \bar{X}_n \tag{3.10}$$

δX_n is a vector of small numbers called the perturbation vector. If $\Phi(t_{n-1} - t_n)$ is applied to both sides of equation 3.10, equation 3.11[3] is obtained. This proves that the polynomial transition matrix $\Phi(\zeta)$ can be applied to the perturbation vector to retrodict it to any time instance, and so equation 3.12[3] can be written.

$$\Phi(t_{n-1} - t_n)\delta X_n = \Phi(t_{n-1} - t_n)(X_n - \bar{X}_n) = X_{n-1} - \bar{X}_{n-1} = \delta X_{n-1} \tag{3.11}$$

$$\delta X_{n-1} = \Phi(t_{n-1} - t_n)\,\delta X_n \tag{3.12}$$

Equation 3.12 will be needed to build the T matrix. First the observation sensitivity matrix needs to be developed.

3.3.2 The observation sensitivity matrix

The nominal state vector $\bar{X}_n$ can be used to generate a simulated observation vector, as in equation 3.13 [3].

$$\bar{Y}_n = G(\bar{X}_n) \tag{3.13}$$

$\bar{Y}_n$is the set of observations that would be seen in the absence of noise at time t_n if $\bar{X}_n$was the real life state vector of the aircraft being tracked. In this particular case the observation equation G is defined in section 3.2.

The difference between the actual and simulated observation vectors is the observation perturbation vector and can be developed using formula 3.14 [3].

$$\delta Y_n = Y_n - \bar{Y}_n = G(X_n) - G(\bar{X}_n) + N_n \tag{3.14}$$

Rearranging equation 3.10 to make X_n the subject of the formula and then substituting in 3.14 gives equation 3.15 [3].

$$\delta Y_n = G(\bar{X}_n + \delta X_n) - G(\bar{X}_n) + N_n \tag{3.15}$$

In Appendix A, the local linearization process presented by Morrison [3] is described for the perturbation vector δX. This process can be applied to equation 3.15 in a similar fashion to produce the linear form of equation 3.16 [3].

$$\delta Y_n = M(\bar{X}_n)\,\delta X_n + N_n \tag{3.16}$$

In the preceding equation the matrix $M(\bar{X}_n)$ is known as the observation sensitivity matrix [3]. $G(X_n)$ is a p-vector of functions on the elements of $X_n$which is a q-vector. The observation sensitivity matrix is defined in equation 3.17 [3].

$$[M(\bar{X}_n)]_{i,j} = \partial g_i(x_1 \ldots x_q)/\partial x_j \,|_{\bar{X}_n} \quad i = 1\ldots p, \qquad j = 1\ldots q, \tag{3.17}$$

This states that the i, j^{th} element of the observation sensitivity matrix is calculated by taking the derivative of the ith function of G with respect to the jth element of X, evaluated using the elements of the $\bar{X}_n$.

The observation sensitivity matrix described above is valid for one observation instant. Each of the eight receivers produces three observations variables. These are Doppler (f_d), and bearing angle which is split into cosine and sine ($\cos(\psi)$ and $\sin(\psi)$). The state vector consists of the 0th to 10th derivative of the Cartesian co-ordinates x, y and z. Therefore $p = 3$, and $q = 33$. The Doppler observation equation contains only the 0th and 1st derivatives of x, y and z, therefore partial derivatives with respect to all higher order terms of the state vector are equal to 0. The two bearing angle equations only contain the 0th derivatives of x and y, so all higher order partials and all partials with respect to the derivatives of z are again 0. Using equation 3.17 the observation sensitivity matrix is as shown in equation 3.18 [3].

$$M(\bar{X}_n) =$$

$$\begin{pmatrix} D_x f_d & D_{\dot{x}} f_d & 0\ldots0 & \| & D_y f_d & D_{\dot{y}} f_d & 0\ldots0 & \| & D_z f_d & D_{\dot{z}} f_d & 0\ldots0 \\ & & & \| & & & & \| & & & \\ D_x\cos(\psi) & 0 & 0\ldots0 & \| & D_y\cos(\psi) & 0 & 0\ldots0 & \| & 0 & 0 & 0\ldots0 \\ & & & \| & & & & \| & & & \\ D_x\sin(\psi) & 0 & 0\ldots0 & \| & D_y\sin(\psi) & 0 & 0\ldots0 & \| & 0 & 0 & 0\ldots0 \end{pmatrix}_{\bar{X}_n} \tag{3.18}$$

The partial derivatives shown in equation 3.18 are described in Appendix B, and all partials are evaluated using the elements of $\bar{X}_n$. The observation sensitivity matrix is what is used to calculate the total observation matrix, or T matrix for our radar system.

Note that since there are eight receivers, each of the terms in the sensitivity matrix of equation 3.18 is a column vector of eight values.

For example: $D_x f_d$ is the partial derivative of the Doppler observation function with respect to the state vector variable x . This derivative contains constants x_k, y_k and z_k. These are the co-ordinates of the receiver in question. The calculation

is performed for each receiver and the eight value column vector $D_x f_d$ is built using the eight results. Therefore the observation sensitivity matrix has dimensions 24x33.

Having described how to calculate the observation sensitivity matrix the T matrix can now be described.

3.3.3 The total observation matrix calculation

Equation 3.16 is restated:

$$\delta Y_n = M(\bar{X}_n)\,\delta X_n + N_n \tag{3.19}$$

By concatenation of equation 3.19 for all the observation instances equation 3.20 is obtained.

$$\begin{pmatrix} \delta Y_n \\ --- \\ \delta Y_{n-1} \\ --- \\ \vdots \\ --- \\ \delta Y_{n-L} \end{pmatrix} = \begin{pmatrix} M(\bar{X}_n)\delta X_n \\ --------- \\ M(\bar{X}_{n-1})\delta X_{n-1} \\ --------- \\ \vdots \\ --------- \\ M(\bar{X}_{n-L})\delta X_{n-L} \end{pmatrix} + \begin{pmatrix} N_n \\ --- \\ N_{n-1} \\ --- \\ \vdots \\ --- \\ N_{n-L} \end{pmatrix} \tag{3.20}$$

Using the initial proof that gave us equation 3.12, equation 3.20 can be rewritten as equation 3.21.

$$\begin{pmatrix} \delta Y_n \\ --- \\ \delta Y_{n-1} \\ --- \\ \vdots \\ --- \\ \delta Y_{n-L} \end{pmatrix} = \begin{pmatrix} M(\bar{X}_n) \\ ------------ \\ M(\bar{X}_{n-1})\Phi(t_{n-1}-t_n) \\ ------------ \\ \vdots \\ ------------ \\ M(\bar{X}_{n-L})\Phi(t_{n-L}-t_n) \end{pmatrix} \delta X_n + \begin{pmatrix} N_n \\ --- \\ N_{n-1} \\ --- \\ \vdots \\ --- \\ N_{n-L} \end{pmatrix} \tag{3.21}$$

The matrix multiplying δX_n on the right hand side of equation 3.21 is the total observation matrix $T_{(n)}$. In this implementation the delay between observation instances is constant at 0.25s (the integration time "τ" for the cross-correlation function of the radar generator). Therefore $T_{(n)}$ can be written as in equation 3.22.

$$T_{(n)} = \begin{pmatrix} M(\bar{X}_n) \\ ------------ \\ M(\bar{X}_{n-1})\Phi(-\tau) \\ ------------ \\ \vdots \\ ------------ \\ M(\bar{X}_{n-L})\Phi(-L\tau) \end{pmatrix} \tag{3.22}$$

With this definition equation 3.21 can be rewritten as follows:

$$\delta Y_{(n)} = T_{(n)}\delta X_n + N_{(n)} \tag{3.23}$$

Equation 3.23 leads to the central concept of Gauss-Newton filtering, the minimum variance rule.

3.4 The Minimum Variance Rule

The minimum variance rule is a law of estimation theory that determines the most accurate possible estimate.

3.4.1 The residuals

The total observation equation can be rewritten as follows:[3]

$$Y_{(n)} = T_{(n)}X_{(n)} + N_{(n)} \tag{3.24}$$

Recalling the perturbation vector equation:

$$\delta X_n = X_n - \bar{X}_n \tag{3.25}$$

δX_n is the difference between the true state vector and the nominal state vector $\bar{X}_n$. The goal is to attain an estimate of X_n that is more accurate than $\bar{X}_n$. This estimate is given the notation $X^*_{n,n}$. Here the first subscript implies that the estimate is valid at time instant t_n, and the second subscript indicates that it is based on observations up to time instant n. Equation 3.24 can be rewritten as shown in formula 3.26 [3].

$$Y_{(n)} = T_{(n)}X^*_{(n,n)} + N^*_{(n)} \tag{3.26}$$

Rearranging equation 3.26 to make $N^*_{(n)}$ the subject of the formula gives equation 3.27.

$$N^*_{(n)} = Y_{(n)} - T_{(n)}X^*_{(n,n)} \tag{3.27}$$

$N^*_{(n)}$ is the residual vector and $T_{(n)}X^*_{(n,n)}$ is the fitted observation vector. Equation 3.27 is depicted in Figure 3.1.

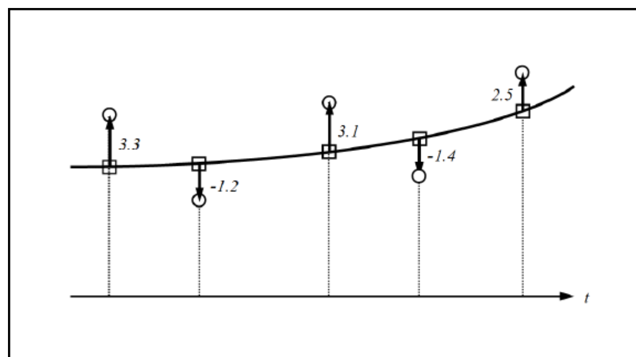

Figure 3.1: The residuals of the observation and fitted observation vectors[3]

The solid line shows the trajectory described by $X^*_{(n,n)}$, the small squares show the fitted observation vector $T_{(n)}X^*_{(n,n)}$ and the circles show the observations $Y_{(n)}$. The arrows depict the residuals $Y_{(n)} - T_{(n)}X^*_{(n,n)}$. If the trajectory shown in Figure 3.1 is considered, the residual vector is:

$$N^*_{(n)} = Y_{(n)} - T_{(n)}X^*_{(n,n)} = (2.5,\ -1,4,\ 3.1,\ -1.2\ 3.3)^T \tag{3.28}$$

Gauss made the profound observation that if the residuals can be weighted according to how accurate they are, the result could perhaps be used to gauge how accurate the estimate $X^*_{n,n}$ is [3]. Gauss measured the accuracy of the residuals using the known variances of the errors in the observation vector, which make up the covariance matrix $R_{(n)}$ [3]. He assumed that the observation errors were uncorrelated, both stage-wise and locally, therefore his covariance matrices were always diagonal [3]. The covariance matrix of the observation vector is given by:

$$R_{(n)} \equiv E\left(N_n^T N_n\right) \tag{3.29}$$

If the residual vector is multiplied by its transpose the result is the sum of the square residuals. Gauss divided each of the square residuals by the known variance of the error of the observation associated with it, to form the sum of the weighted squared residuals show in equation 3.30 [3].

$$\begin{aligned} N^{*T}_{(n)} N^*_{(n)} &= (Y_{(n)} - T_{(n)}X^*_{n,n})^T R^{-1}_{(n)} (Y_{(n)} - T_{(n)}X^*_{n,n}) \\ &= {}^{2.5^2}/\sigma^2 + {}^{1.4^2}/\sigma^2 + {}^{3.1^2}/\sigma^2 + {}^{1.2^2}/\sigma^2 + {}^{3.3^2}/\sigma^2 \end{aligned} \tag{3.30}$$

Since more accurate observation parameters have smaller variances, the most accurate observation in equation 3.30 will have more weight in the sum of residuals calculation, from which the estimate is derived [3]. Gauss realised that the $X^*_{n,n}$ that minimizes equation 3.30 will be the most accurate estimate of X_n that the total observation vector $Y_{(n)}$ is capable of producing [3]. The notation in equation 3.31 [3] is used to define the sum of the weighted squared residuals of $X^*_{n,n}$.

$$e(X^*_{n,n}) \equiv N^{*T}_{(n)} N^*_{(n)} = (Y_{(n)} - T_{(n)}X^*_{n,n})^T R^{-1}_{(n)} (Y_{(n)} - T_{(n)}X^*_{n,n}) \tag{3.31}$$

3.4.2 Derivation of the minimum variance algorithm

The goal is to attain the best possible $X^*_{n,n}$ given the observation vector $Y_{(n)}$ which has known covariance matrix $R_{(n)}$, as well as the total observation matrix $T_{(n)}$ (given a nominal state vector $\bar{X}_{(n)}$). Morrison [3] derives the minimum variance algorithm that estimates the best $X^*_{n,n}$ as follows. The definition of the sum of the weighted squared residuals is formed in equation 3.31. $e(X^*_{n,n})$ is a quadratic form on the positive definite matrix $R^{-1}_{(n)}$ and so is always positive [3]. Figure 3.2 shows $e(X*_{n,n})$ as a surface which, when traversed, represents all the possible values of $X^*_{n,n}$.

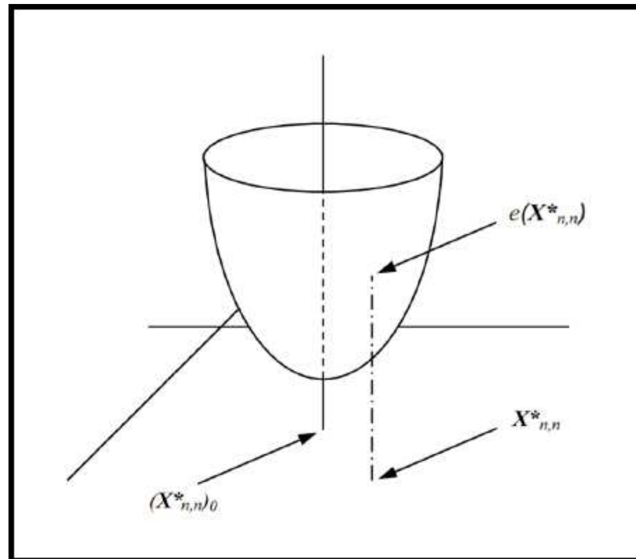

Figure 3.2: Sum of weighted squared residuals as a surface[3]

If equation 3.31 is differentiated with respect to each of the elements of $X^*_{n,n}$, and the result set equal to $\oslash$, $e(X^*_{n,n})$ is minimized, and therefore a set of equations are provided that allow the best estimate of $X^*_{n,n}$ to be attained. The result as found by Morrison [3] is shown in equation 3.32, and is explained further in Appendix C.

$$X^*_{n,n} = \left(T^T_{(n)} R^{-1}_{(n)} T_{(n)}\right)^{-1} T^T_{(n)} R^{-1}_{(n)} Y_{(n)} \tag{3.32}$$

If the total observation equation (3.23) is restated as equation 3.33, the above differentiation procedure can be repeated to formulate equation 3.34.

$$\delta Y_{(n)} = T_{(n)}\,\delta X^*_{n,n} + N^*_{(n)} \tag{3.33}$$

$$\delta X^*_{n,n} = \left(T^T_{(n)} R^{-1}_{(n)} T_{(n)}\right)^{-1} T^T_{(n)} R^{-1}_{(n)}\,\delta Y_{(n)} \tag{3.34}$$

In equation 3.34, δX^*_n is the difference between the nominal state vector and the best estimate $X_{n,n}*$. This equation is what is required; It allows an estimate of the state vector of the aircraft to be made from a set of known inputs. If $\delta X^*_{n,n}$ is added to $\bar{X}_n$ the result should be the best estimate $X^*_{n,n}$, however for various reasons this is not the case. Local linearization has to be used in deriving $T_{(n)}$, where the higher terms of the Taylor's series expansion were dropped to avoid non-linearities, resulting in a loss of accuracy since the problem of aircraft trajectory estimation is typically an inherently non-linear one [3]. $\delta X^*_{n,n}$from equation 3.34 is used to get closer to the best estimate that the set of observations can produce, and the result leads to the iterative Gauss-Newton procedure that is used to get even closer.

3.5 The iterative Gauss-Newton algorithm

Since the estimation procedure described so far uses the minimum variance rule, which relies on linearity, when non-linearities are present an iterative procedure is developed in which the estimate from each iteration is used as the input "nominal state vector" to the next one. The minimum variance rule for the estimation procedure gives equation 3.34, in which the only required inputs are the total observation vector $Y_{(n)}$ (from which $\delta Y_{(n)}$ is derived using equation 3.14), the

known covariance matrix $R_{(n)}^{-1}$ and the total observation matrix $T_{(n)}$. $T_{(n)}$ requires the knowledge of a nominal state vector $\bar{X}_{n,n}$, the procedure for deriving which is developed in this section. The output estimation $X_{n,n}^{*}$ of each Gauss-Newton iteration is used to develop the total observation matrix $T_{(n)}$ of the next iteration until the required accuracy threshold known as the stopping rule is reached.

In what follows the initialization of the Gauss-Newton algorithm that produces the nominal state vector $\bar{X}_{n,n}$ is described. Then the stopping rule that is used to determine whether or not the state vector estimation produced is of sufficient accuracy is outlined.

3.5.1 Initialization of the nominal state vector

The state vector used consisted of the 0th up to the 10th derivatives of x, y and z. The Gauss-Newton method is so robust that it can be initialized with estimates of just the 0th and 1st derivatives of the x and y co-ordinates. The method used to estimate position was to average the crossing points of the bearing lines from the set of receivers. Weak crossing points that would throw the estimate off were eliminated using the angle between the bearing lines [3]. The second derivative was obtained using an EMP filter for each of the co-ordinates (x and y). The method is covered in detail by Morrison [4]. The EMP filter used in this implementation was a 1st degree current estimate EMP filter. The filter finds the straight line (degree 1 polynomial) best fit to the radar data in terms of least squares, has an expanding memory and produces estimates up to and based on the current time instant.

A block diagram of the algorithm is shown in Figure 3.3.

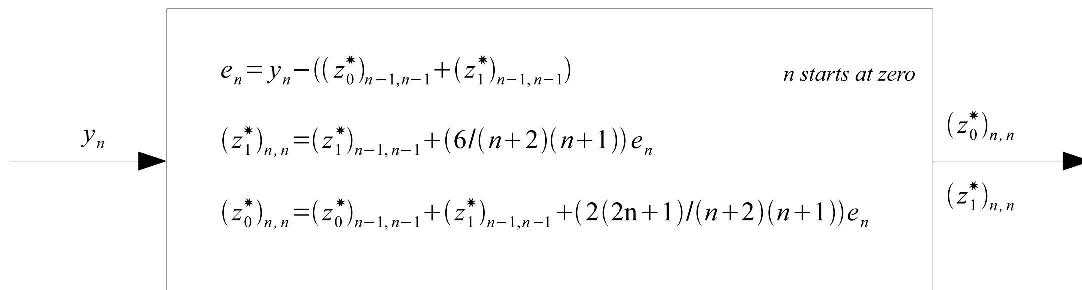

Figure 3.3: 1st degree current estimate EMP algorithm[3]

In this diagram y_n is a vector of inputs valid up to time instant t_n. The outputs $(z_0^*)_{n,n}$ and $(z_1^*)_{n,n}$ are the 0th and 1st derivatives of the smoothed data, valid at time instant t_n. The algorithm is recursive, and starts with $n = 0$. Since the algorithm is self initializing the values of $(z_0^*)_{n,n}$ and $(z_1^*)_{n,n}$ when $n = 0$ are not important, and might as well be set to 0.[3].

3.5.2 The stopping rule for Gauss-Newton iteration

In order to determine when to cease iteration of our Gauss-Newton algorithm, three simple tests are used. These are described below:

Maximum iterations limit

This rule places an upper bound on the number of iterations that are allowed per state vector estimation, and simply terminates when the bound is reached. In this implementation it is used only when the other two tests fail.

Perturbation vector limit

This rule puts a limit on the quantity that is being driven toward zero, the perturbation vector $\delta X^*_{n,n}$. The test is shown in equation 3.35 [3].

$$is \qquad \|\delta X\| < \varepsilon \,? \tag{3.35}$$

Since the vector $\delta X^*_{n,n}$ contains values with different units (m, ms^{-1} etc) they must be normalized to avoid higher magnitude terms from dominating the calculation. This can be done by dividing each element by the square root of the corresponding diagonal element of the covariance matrix $S^*_{n,n} = \left(T^T_{(n)} R^{-1}_{(n)} T\right)^{-1}$ which approximate the variances of the elements of $\delta X^*_{n,n}$ [3].

Successive estimation difference limit

As the output of the iterative algorithm converges towards the most accurate estimate, the difference between successive estimations can be expected to get smaller and smaller. The difference between successive estimates can therefore be used as a stopping rule. This is illustrated by equation 3.36, where k is the current estimation.

$$is \qquad \|(X^*_{n,n})_k - (X^*_{n,n})_{k-1}\| < \varepsilon \,? \tag{3.36}$$

In equation 3.36 the same method of normalization is used as in equation 3.35.

Chapter 4

IDL implementation profiling

The profiling of the original IDL code on the base platform is outlined in this chapter. The radar data generator used to provide data for the algorithm is briefly introduced, the flow of data through the simulator is outlined, and the actual profiling results are then presented.

The pre-existing IDL implementation, shown in Figure 4.1, can be thought of in terms of components external to the tracking algorithm, and the tracking algorithm itself. The external components are referred to as the external model, which simulates the real world environment that would provide the algorithm with data in the field. The external model is represented by the flight path generator and radar data generator in Figure 4.1. The properties of the external model are controlled by the parameter selection interface. Note that the radar data generator used in the simulator is fairly simplistic, and the Gauss-Newton implementation under investigation has yet to be tested under more realistic conditions.

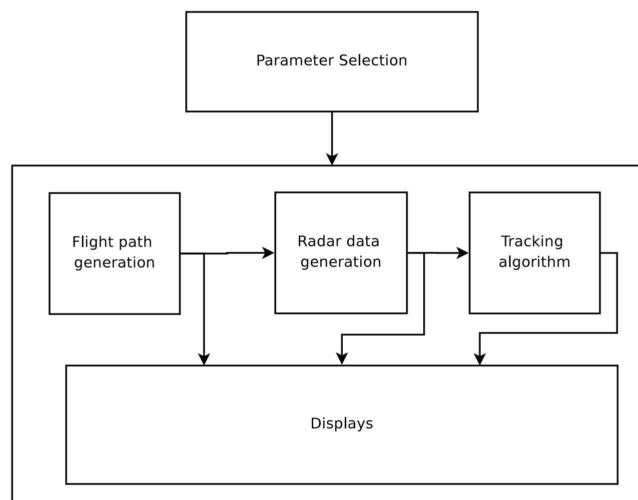

Figure 4.1: Block diagram of pre-existing PCL simulator, as written by Dr Richard Lord in the IDL programming language, which was used to test the Gauss-Newton algorithm[2]

What is of interest here is the tracking algorithm block, which implements the filter model based on input data from the external model, and that will therefore be the main focus of the discussion. However a brief description of the radar

data generator is also included to provide a clear understanding of the flow of data into the filter. The tracking algorithm consists of three major components:

- The Master Control Algorithm (MCA), which controls the flow of data through the other components.
- The track initialization component, which provides a nominal state vector $\bar{X}_{n,n}$ for input to the Gauss-Newton algorithm
- The Gauss-Newton tracking algorithm

4.1 Radar data generation

The simulated PCL radar system used a continuous wave transmitter radiating at a frequency of 600MHz ($\lambda = 0.499654m$) located at the origin of a Cartesian co-ordinate system. The receiver network is set up such that 8 approximately evenly spaced receivers are located around the edge of a circle of radius 40km. The integration time "τ" for the cross-correlation function of the radar generator was 0.25s, resulting in 4 observation samples per second. Each sample consisted of a Doppler and Bearing reading from each of the 8 receivers.

The landing passenger aircraft tracking simulation involves 800 observation samples, made at a rate of 4 samples a second, thus the flight time is 200 seconds. The first set of observation data was sent to the tracking algorithm after 20 seconds, and thus consisted of 80 samples. Subsequent cycles, or observation instances, occurred every 5 seconds, and consisted of 80 samples (20 new observations and the last 60 from the previous observation instant).

The MCA takes in the observations as input from the radar data generator. The simulator generates a fresh set of observations (and corresponding simulated flight path) for each simulation run (Figure 4.2 shows one simulation run). Note that for this implementation the same set of receivers, radar and aircraft parameters were used for each simulation run.

Profiling was carried out by gradually breaking down each functional block into logical elemental components in a top down fashion, starting with the three major components just mentioned. To time the execution of functional blocks of code through calls were made to the system clock. The platform specifications are shown in Table 4.1. In what follows the MCA and the Gauss-Newton algorithm profiling are described.

Table 4.1: Base platform specification

CPU family	Intel(R) Xeon(TM)
CPU clock speed	4 x 3.00GHz dual core (x84_64) with 2MB cache
Memory	2GB
Storage	2x 300GB SATA HDD
Co-processor interface	PCI-X 2.0 133 MHz
Operating System	SuSE Linux Enterprise Server9 (SLES9)

4.2 Master Control Algorithm profiling

The flow of control of the MCA is shown in Figure 4.2. Each block of the figure is considered individually in the discussion that follows. The input to the MCA is the vector of radar data observations for an entire simulation.

First block: $Y_{(n)} = \mathbf{Y}_{(N-L\,:\,N)}$

Let the vector $\mathbf{Y}$ be a vector of 800 observations for an entire simulation run, as generated by the external model. The Gauss-Newton algorithm, under control of the MCA, cycles for the first time after 20 seconds (after 80 observations have been made), and thereafter every 5 seconds. Therefore 20 new observation samples are input to the algorithm every cycle or observation instant. The memory length used for the observations on each cycle is 80 (20 new observations, and the last 60 from the previous cycle). Thus the most recent 80 observations are used in each cycle. The subscript N represents the most recent sample, where one new sample occurs every 0.25 seconds. The variable L is used to represent the number of new observations per cycle, thus $L = 20$. On entry to the flow chart $N = 80$, and 20 is added to N at the start of every cycle. Thus the notation $\mathbf{Y}_{(N-L\,:\,N)}$ in the first block of the MCA flow chart represents the set of observation samples used for each cycle, $Y_{(n)}$.

Second block: Init Track

The input to track initialization is the vector of bearings $\Psi_{(n)}$ (extracted from $Y_{(n)}$), and the output is the nominal state vector $\bar{X}_{(n)}$.

Third block: Gauss-Newton

The input to the Gauss-Newton algorithm is the covariance matrix of our observation vector $R^{-1}_{(n)}$, the total observation vector $Y_{(n)}$ and the nominal state vector $\bar{X}_{(n)}$, and the output is an estimate of the state vector $X^*_{(n)}$. Note that the Gauss Newton algorithm has knowledge of the total state vector transition matrix $\Phi_{(n)}$.

Final block: GOF

The Goodness Of Fit (GOF) decision block is based on the minimum average of the absolute values of the perturbation vector δX^*_n, the minimum observation residuals (the square roots of the sum of squares of the observation perturbation vectors $\delta\Psi_{(n)}$ and $\delta f_{d(n)}$) and the maximum iterations limit (50). If none of the GOF criteria are met, then the nominal state vector for the next iteration is set to the state vector estimation ($\bar{X}_{(n)}$ is set to $X^*_{(n)}$) and Gauss-Newton repeats. If the GOF criteria are met then $X^*_{(n)}$ from the Gauss-Newton iteration is the best estimate of the state vector $X^*_{(n)_{final}}$ and becomes the output of the current cycle. If there are further observations in $\mathbf{Y}$ then the MCA cycles again.

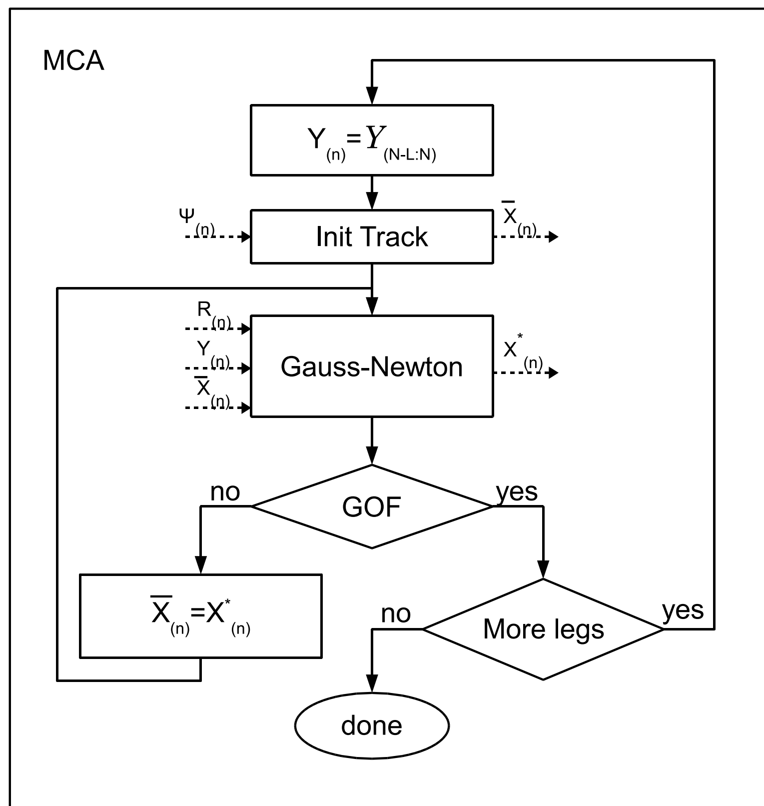

Figure 4.2: Master Control Algorithm flow diagram for the IDL implementation. The notation GOF stands for Goodness Of Fit and symbolizes the stopping rule discussed in chapter 3. The decision block labeled *More legs* simply determines whether there are more observation instances remaining in the simulation. If so the simulator cycles again, if not the simulation ends.

In profiling the MCA times were averaged over 10 simulations, each with a different flightpath as input. Each simulation consists of 37 observation instances, therefore times for the main functional blocks are averaged over 370 readings. The results of the MCA profiling are shown in bar chart of Figure 4.3. As expected the Gauss-Newton algorithm accounts for about 94% of the execution time. As Gauss-Newton dominates the execution time and track initialization is a fairly straight forward process, only the Gauss-Newton block was broken down further.

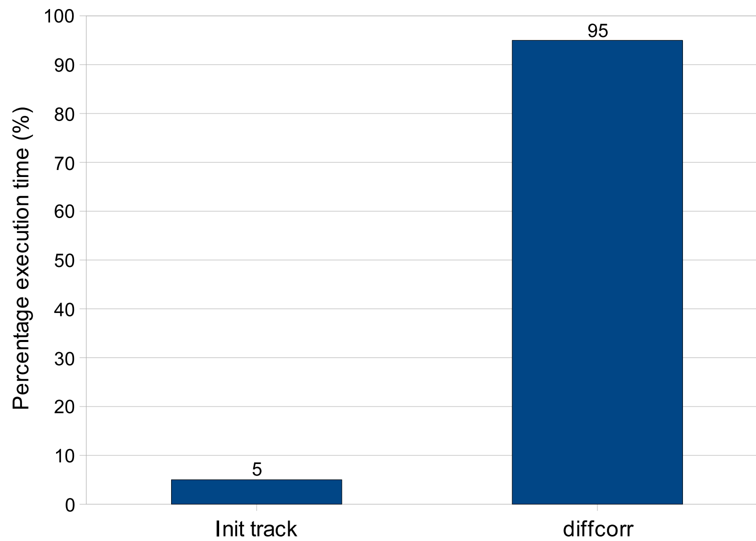

Figure 4.3: Profiling of MCA showing percentage execution time

4.3 Gauss-Newton method profiling

The operation of the Gauss-Newton method is described here:

1. Assemble the total simulated observation vector $\bar{Y}_{(n)}$ (using $\bar{X}_{(n)}$), and the observation perturbation vector $\delta Y_{(n)}$ ($\delta Y_{(n)} = Y_{(n)} - \bar{Y}_{(n)}$) as well as the RMS residuals of the observations.
2. Assemble the total observation matrix $T_{(n)}$. This is done in a loop that iterates through the observation samples of the observation instance. Let the iterator i represent a single time instance. The stages of each iteration are:
   (a) Calculate the sensitivity matrix $M(X_i)$ using the partial derivatives of the observation equations with respect to the elements of the state vector evaluated at $\bar{X}_i$.
   (b) Calculate a few rows of the total observation matrix $T_{(n)}$ by multiplying the sensitivity matrix by the transition matrix for the sample, Φ_i.
3. Transpose $T_{(n)}$ then multiply by $R^{-1}_{(n)}$ giving $T^T_{(n)}R^{-1}_{(n)}$.
4. Generate covariance matrix by computing $T^T_{(n)}R^{-1}_{(n)}T_{(n)}$, then inverting it to form $(T^T_{(n)}R^{-1}_{(n)}T_{(n)})^{-1}$.
5. Compute $\delta X_n = (T^T_{(n)}R^{-1}_{(n)}T_{(n)})^{-1}T^T_{(n)}R^{-1}_{(n)}\delta Y_{(n)}$.
6. Generate new state vector estimate for the most recent time instant: $(\bar{X}_n)_{new} = (\bar{X}_n)_{old} + \delta X_n$then build $(\bar{X}_{(n)})_{new}$ using the transition matrices$\Phi_{(n)}$and $(\bar{X}_{(n)})_{new}$
7. The algorithm then tests for convergence (or goodness of fit) using the RMS residuals, average of δX_n, and number of iterations that have occurred[1]. If the test is negative differential correction iterates again.

[1]The maximum number of iterations used in profiling was 50. This maximum has been found to be sufficient [citation], and indeed was never reached during the course of the profiling.

Conclusions

The results of the Gauss-Newton algorithm profiling are shown in the graph of Figure 4.4. The main areas of computation are:

- Observation perturbation vector loop (1 above). This is labeled *dY* in the figure and accounts for about 9% of the execution time.
- Total observation matrix loop:
  - Sensitivity matrices calculation (2a above). Labeled *partials*, accounts for 26.5%.
  - Matrix multiplication to calculate $T_{(n)}$ rows (2b above). Labeled *T rows*, accounts for 27.7%. This includes matrix transpose and scaling to calculate $T_{(n)}^T R_{(n)}^{-1}$ (3 above), which accounts for 6%.
- Matrix multiplication and inversion to form covariance matrix (4 above). Labeled *CovM*at, accounts for 12%.
- Matrix multiplications in perturbation vector calculation (5 above). Labeled *dX*, accounts for 21%.
- Retrodiction of state vector to all time instance. Labeled *retrodict*, accounts for 2.79%.

These cumulatively account for 96% of the Gauss-Newton algorithm execution time.

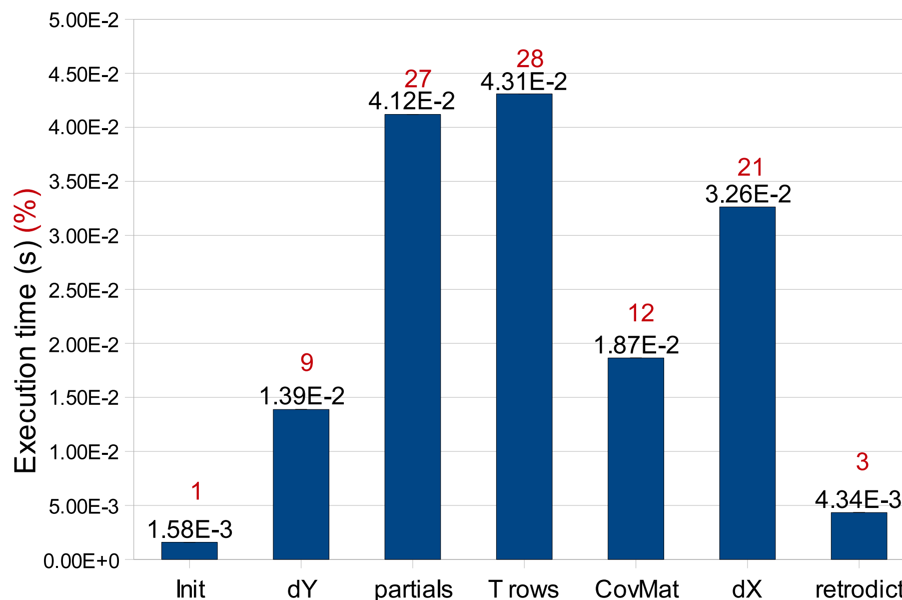

Figure 4.4: Graph illustrating IDL implementation of Gauss-Newton profiling. The observation perturbation vector loop is labeled *dY*. The partial derivatives calculation for the sensitivity matrix is labeled *partials*. The multiplications of the sensitivity matrices with the transition matrices is labeled *Trows*. This *Trows* execution block includes the observation matrix transpose and scaling with the error vector covariance matrix. Calculation of the covariance matrix $(T_{(n)}^T\ R_{(n)}^{-1}\ T_{(n)})^{-1}$ is labeled *covMat*. The multiplication of the covariance matrix $T_{(n)}^T\ R_{(n)}^{-1}$ and the observation observation perturbation vector to form the state vector perturbation vector is labeled *dX*. The addition of the perturbation vector to the previous state vector estimate (or nominal state vector), and the retrodiction of the new state vector estimate to all time instances using the transition matrix is labeled *retrodict*.

The profiling of the IDL implementation reveal that the most computationally expensive areas of the algorithm are the arithmetic operations in calculating the partial derivatives of the observation functions, and the high dimension matrix multiplications. The partial derivative calculations consist of a large concentration of double precision arithmetic, repeated in a loop.

Chapter 5

Implementation of Gauss-Newton tracking in C

This section outlines the methodology used to port the Gauss-Newton filtering algorithm to the C programming language and the results of profiling and accuracy testing.

5.1 Description of Implementation

Input from the original IDL external model is used, and the output was tested using IDL code adapted from the original simulator. This is illustrated by Figure 5.1. The flightpath and radar data generator from the IDL simulator was used to write the simulated (external model or "actual") flightpath and the associated set of radar observations to file. These radar observations were read by the C program, and an implementation of the tracking algorithms used to develop an approximation to the flightpath using Gauss-Newton. The C implementation wrote the approximated flightpath to file, which was then read by another IDL program (again adapted from the original simulator) along with the simulated actual flight path, which were compared in accuracy testing.

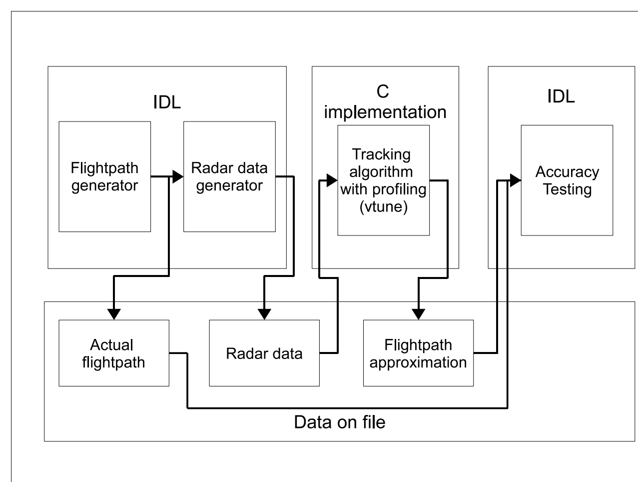

Figure 5.1: Block diagram of C implementation of algorithm with accuracy testing and profiling

The tracking algorithm implementation in C was similar to the IDL version. The overall functionality is illustrated by Figure 5.2. The main differences between the C and IDL versions, are as follows:

- Some of the syntactical conveniences of IDL such as vector and matrix operations are unavailable in standard C, their implementation was however a trivial programming task.
- Mathematical operations such as sine and "square root" have disparate implementations in C and IDL.
- The matrix multiplications and inversions involved use the BLAS ATLAS library in the C implementation. ATLAS (Automatically Tuned Linear Algebra Subroutines) libraries are optimized for the host processor architecture at build time.

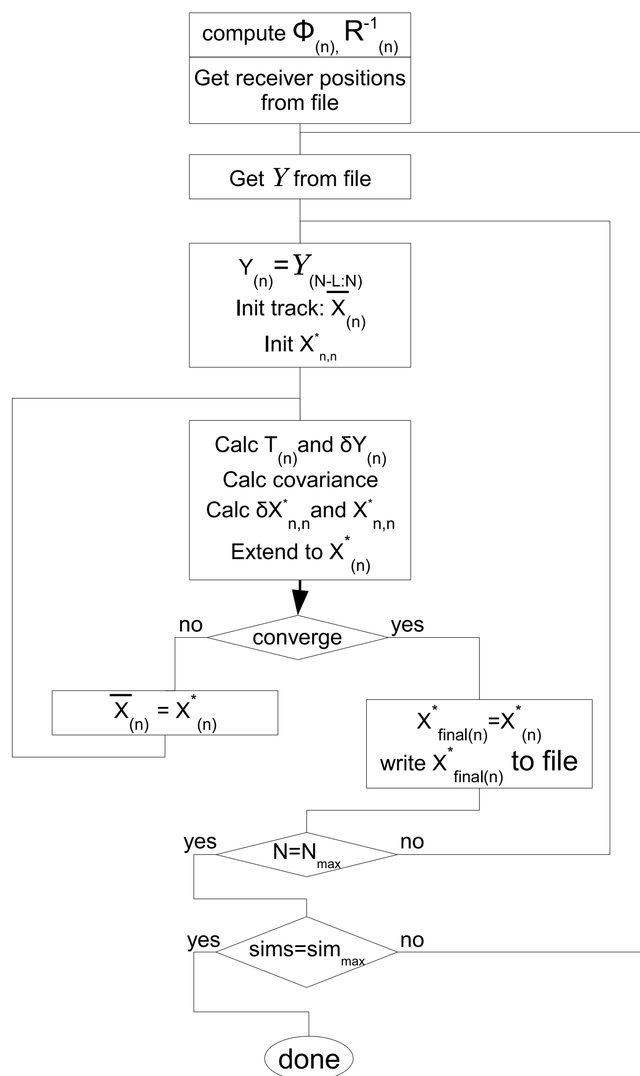

Figure 5.2: C implementation of Gauss-Newton algorithm. For the most part the C version mirrors the original IDL algorithm code, subtle differences can be seen due to the high level nature of IDL which implements programming conveniences such as vector and matrix operations which are unavailable in standard C. The C version utilized the ATLAS BLAS library for the various matrix operations involved in the algorithm.

Figure 5.2 is explained as follows:

- First block: compute the covariance matrix of the total observation vector and the state vector transition matrix, read the receiver positions from file.

- Second block: read the 800 observations for an entire simulation from file.

- Third block: read the total observation vector for the current cycle (observation instant) from 800 observations. Use this to initialize the track (compute the nominal state vector), which is then used to initialize the state vector estimate.

- Fourth block: calculate the total observation matrix and the observation perturbation vector. Use this along with the observation vector covariance matrix to develop the perturbation vector, which is used to refine the state vector estimate. The state vector estimate is then retrodicted to all time instances using the state vector transition matrix.

- Test for convergence: if yes write the state vector estimate to file, otherwise replace the nominal state vector with the current estimate and iterate Gauss-Newton again (return to the fourth block).

- If Gauss-Newton has converged: If the aircraft flight path is not complete (there are more observation instances) then return to the third block and cycle Gauss-Newton again (return to the third block). Otherwise start the next simulation. The number of simulations to run is set as a parameter in the header file.

5.2 C implementation accuracy testing

The flightpath approximation from C was compared with the actual flightpath generated by the IDL simulator using a simple IDL program. The accuracy was tested for 60 simulations, each with a different flightpath. Each simulation consisted of 37 observation instances, thus the accuracy data is taken over 2220 executions of the algorithm. The accuracy of the position and velocity output of the C implementation is illustrated by the graphs of Figure 5.3. These accuracy values compare favourably with those of the original IDL Gauss-Newton implementation.

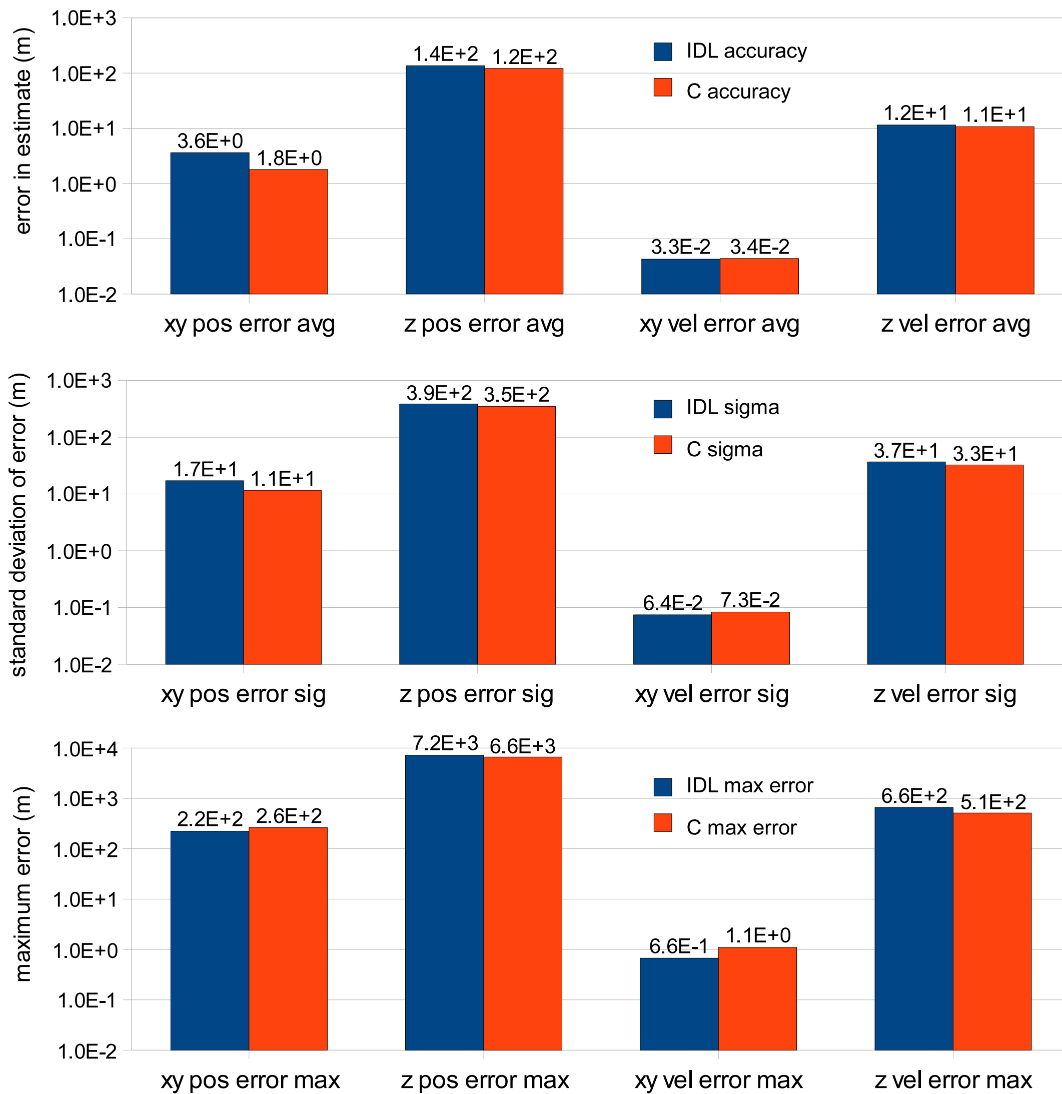

Figure 5.3: Graphs of comparative C and IDL implementation accuracy. The variables shown are position and velocity in the X/Y plane, as well as altitude and its rate of change. The average error over 2220 executions is shown in the top graph, followed by the standard deviation of these errors, and finally the maximum error of each state vector variable in the lower graph. As is evident from the graphs, the accuracy of the C implementation compares favourably with the original IDL version.

5.2.1 Single precision accuracy testing

A version of the C implementation using single instead of double precision for the data structures was also tested. This would have been of benefit when offloading computation to the ClearSpeed board, both in terms of data transfer overhead and the limited memory (particularly poly memory) available on the card. Unfortunately single precision accuracy was found to be insufficient for the Gauss-Newton algorithm. The results of computation in the single precision version were progressively inaccurate as the errors propagated, the final state vector estimates consisting of NaN (Not a Number) values. A NaN is generated when arithmetic operations are performed on infinity and zero (resulting from arithmetic overflow or underflow respectively). It can therefore be concluded that single precision was insufficient for the C implementation.

5.3 C implementation profiling

The results of the C implementation profiling are shown in 5.4. The times shown are in micro-seconds and are for a single Gauss-Newton iteration.

The observation perturbation vector and total observation matrix loops were combined in the C implementation, this is labeled *T matrix* in Figure 5.4.

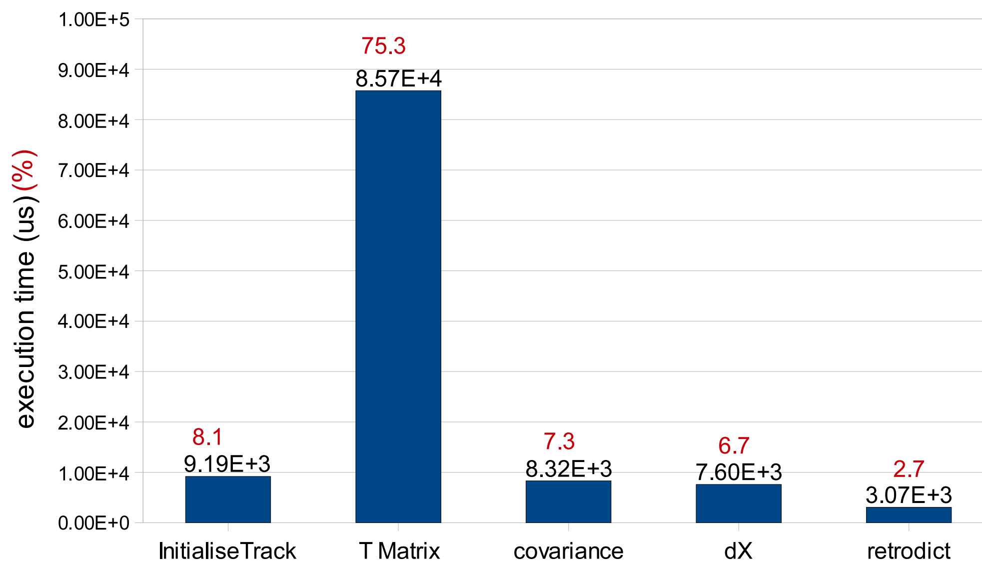

Figure 5.4: Execution times for Gauss-Newton C version. The *T matrix* block consists of the total observation matrix and observation perturbation vector calculations. The *covariance* block is the matrix multiplication and inversion involved in developing the covariance matrix $(T_{(n)}^T R_{(n)}^{-1} T_{(n)})^{-1}$. *dX* signifies the perturbation vector calculation, and *retrodict* the retrodiction of the state vector to all time instances.

Figure 5.5 shows how computation of the total observation matrix is split between the development of the sensitivity matrix $M(\bar{X}_{(n)})$, and the multiplication of $M(\bar{X}_{(n)})$ by Φ. The bar labeled sensitivity also accounts for the calculation of the observation perturbation vector $\delta Y_{(n)}$.

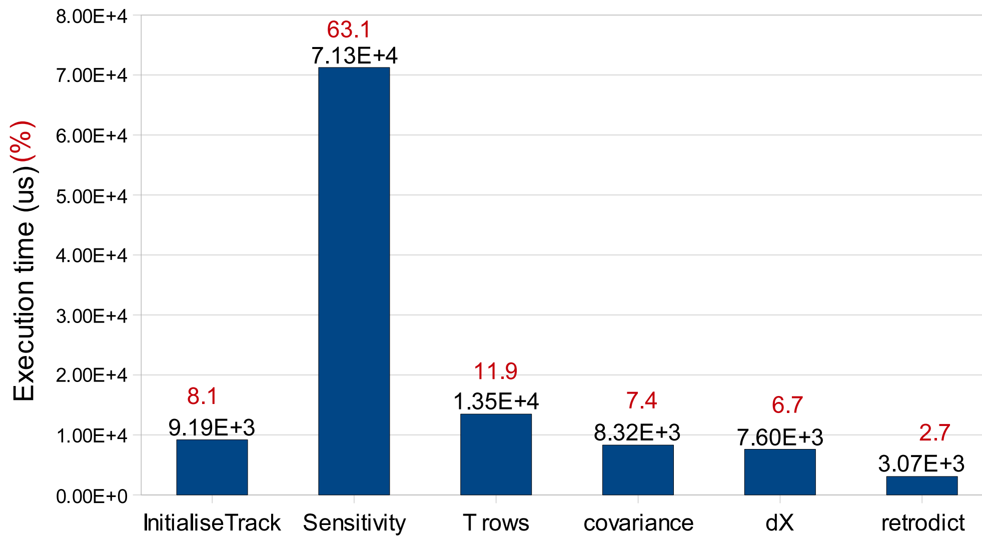

Figure 5.5: Execution times for observation matrix C implementation. The sensitivity block includes the sensitivity and observation perturbation vector calculation.

Figure 5.6 shows a break down of the covariance matrix calculation execution. Computation is split between the matrix multiplication (DGEMM) that produces $T^T_{(n)}R^{-1}_{(n)}T_{(n)}$ and the inversion of that matrix. DGETRF performs the LU decomposition that produces the pivot vector for the matrix inversion, while DGETRF performs the actual inversion.

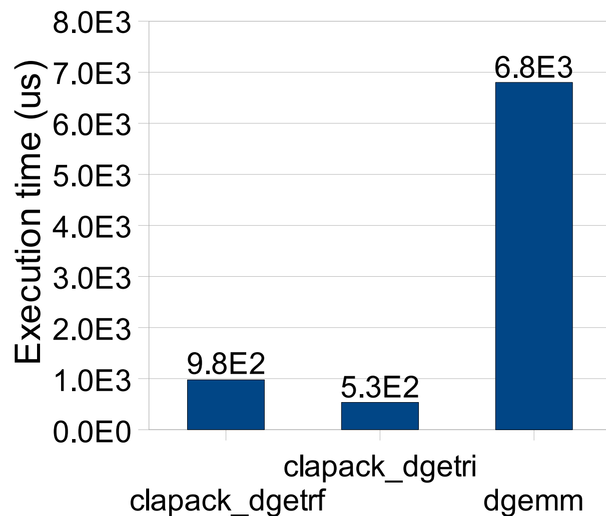

Figure 5.6: Execution times for covariance matrix C version. *dgemm* Performs the multiplication of $T^T_{(n)}R^{-1}_{(n)}$ by $T_{(n)}$. *clapack_dgetrf* and *clapack_dgetri* implements the inversion of the result.

Figure 5.7 shows where the double precision matrix multiplication (DGEMM) execution time is spent. The matrix multiplications in computing the total observation matrix take the longest, followed by the perturbation vector $\delta X_{n,n}$, covariance

matrix (inverse), retrodiction of the state vector and finally multiplications used by the matrix inversion library functions (for the covariance matrix).

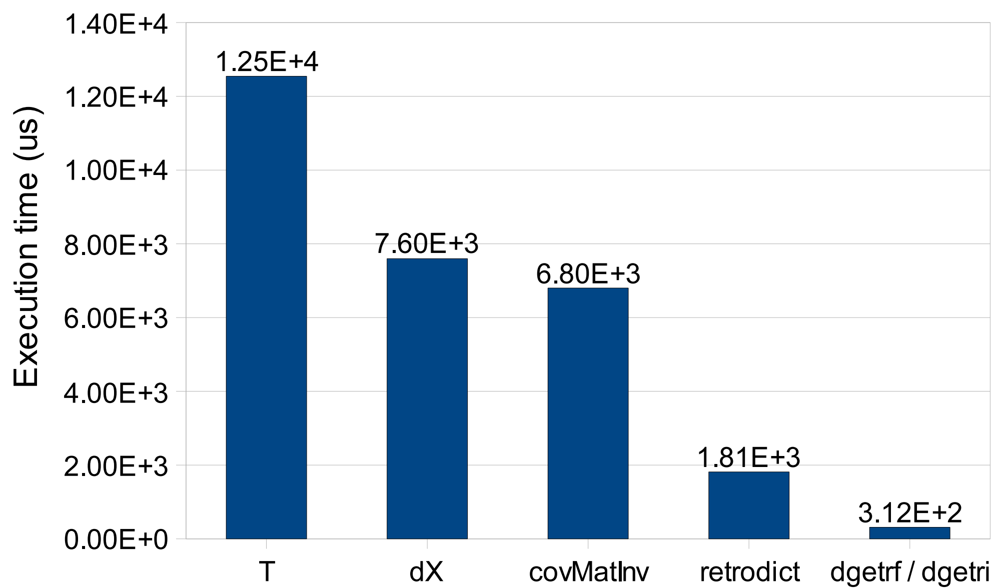

Figure 5.7: Distribution of matrix multiplication execution times for C version

Comparing the percentage execution times of C to IDL shows that in the C implementation matrix multiplication is relatively inexpensive. This can be attributed to the optimized matrix multiplication routines used in the C implementation, which makes use of the ATLAS BLAS library. ATLAS libraries are tuned specifically for optimal performance on the host platform at build time. The C and IDL execution times are compared in Figure 5.8.

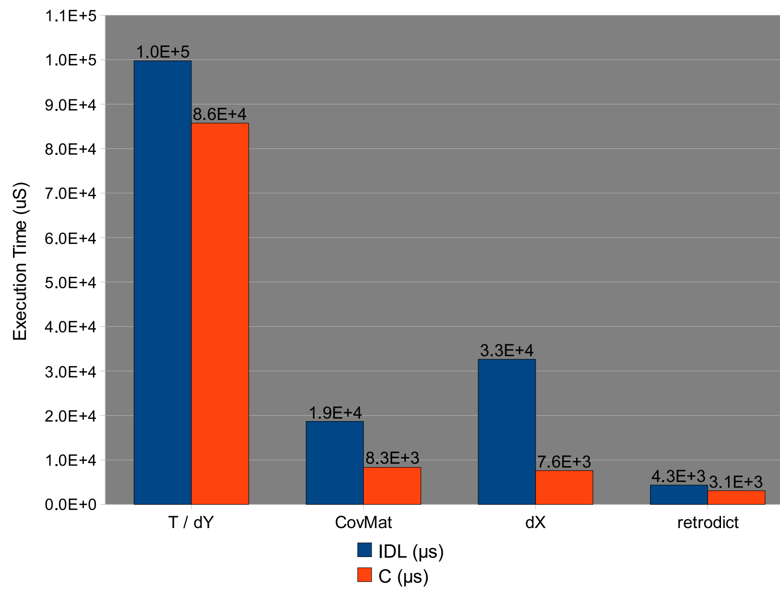

Figure 5.8: Comparative C and IDL profiling graph. Matrix multiplication is relatively inexpensive in the C implementation because of the use of the ATLAS BLAS library which is optimally tuned for performance on the host platform at build time.

Conclusions

The Gauss-Newton algorithm was successfully implemented, with accuracy results that compared favourably with the original IDL implementation. The performance of the C version was 1.38 times faster than the IDL implementation, mainly due to the efficiency of the hardware tuned ATLAS BLAS library.

Chapter 6

Acceleration of the Gauss-Newton algorithm with ClearSpeed

This section contains a description of the investigation and implementation of co-processor offloading in order to accelerate the Gauss-Newton tracking algorithm for PCL. The design stage is detailed, including flow charts and performance prediction. The implementation is then discussed, followed by a description of the code verification and profiling.

6.1 Design of ClearSpeed assisted implementation

The code sections identified as candidates for ClearSpeed acceleration are those that involve unroll-able loops, and the double precision matrix multiplication linear algebra routines. Unroll-able loops exhibit data independence between consecutive iterations, which means the iterations may be processed simultaneously.

The sections of code identified for ClearSpeed acceleration are shown in Figure 6.1 (note that this is a simplification of Figure 5.2). The lines shown in red contain computationally intensive unroll-able loops, and the areas in blue consist of high dimension, double precision matrix multiplication. These sections account for approximately 75% and 16% of the execution time of the algorithm respectively, as shown in section 5.3. The proposed acceleration of each of these areas of code will be outlined in the following sections.

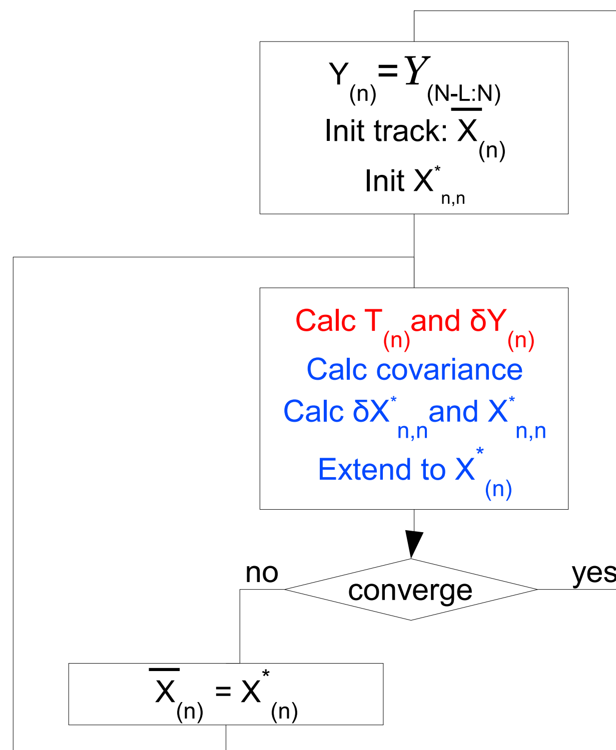

Figure 6.1: Identification of areas of computation as candidates for ClearSpeed acceleration. The lines shown in red contain computationally intensive unroll-able loops, and the areas in blue consist of high dimension, double precision matrix multiplication.

A flowchart of the intended implementation is shown in Figure 6.2. This diagram shows one cycle of the Gauss-Newton algorithm. The observation data used to compute $\bar{X}_{(n)}$ and $\delta Y_{(n)}$ is updated after convergence every cycle as explained in section 4.2, but is omitted from this flowchart for simplicity. In this proposed implementation separate processors on the Advance CSX600 board would be used for the parallel T matrix calculation and the CSXL library calls to avoid overhead in unloading and reloading the T matrix program between iterations.

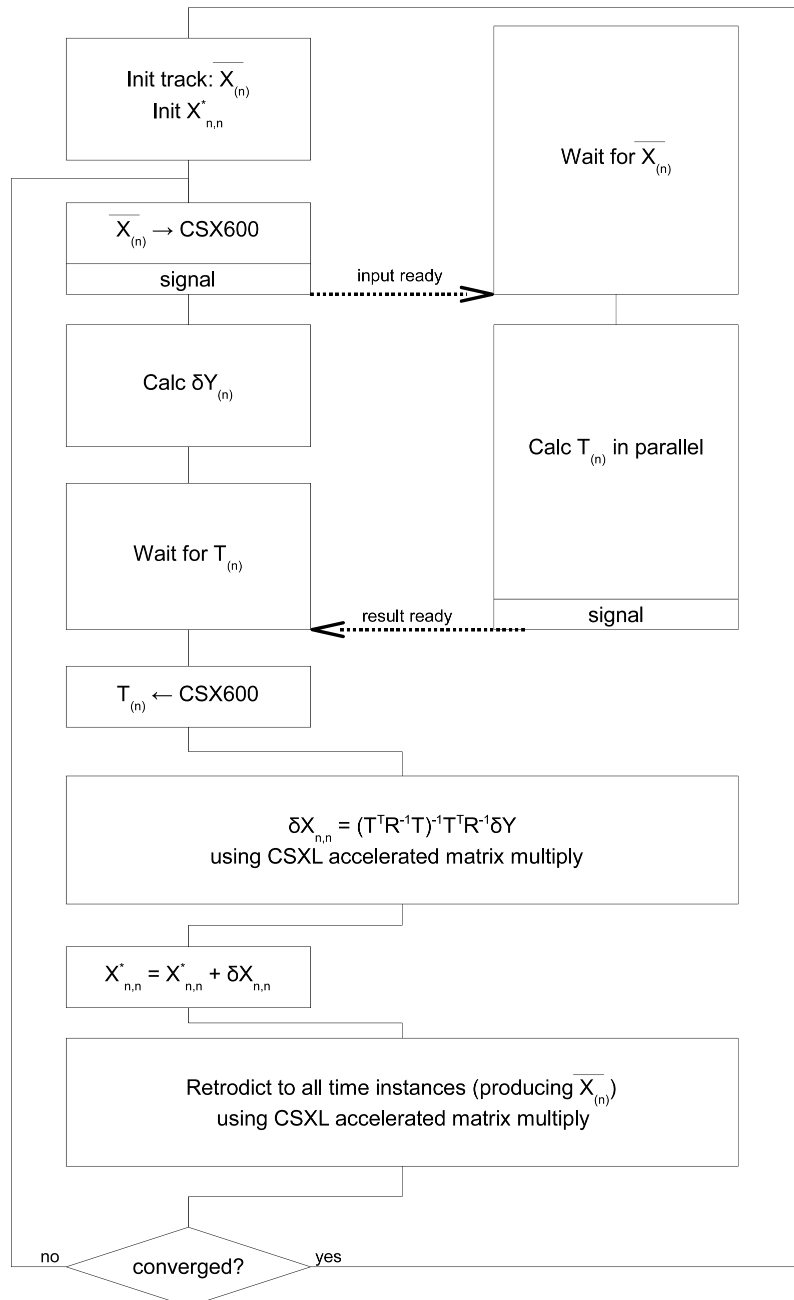

Figure 6.2: Intended ClearSpeed accelerated Gauss Newton implementation flow of control. The sections of code identified as candidates for ClearSpeed acceleration were those involving loops with no data dependency between iterations, and the high dimension double precision matrix multiplications.

6.1.1 Proposed acceleration of the total observation matrix calculation

Unrolling the loop that calculates the total observation matrix $T_{(n)}$ and perturbation vector $Y_{(n)}$ presents a problem. Restating equation 3.22:

$$T_{(n)} = \begin{pmatrix} M(\bar{X}_n) \\ ------------ \\ M(\bar{X}_{n-1})\Phi(-\tau) \\ ------------ \\ \vdots \\ ------------ \\ M(\bar{X}_{n-L})\Phi(-L\tau) \end{pmatrix} \quad (6.1)$$

$M(\bar{X}_n)$ is the sensitivity matrix for one time instant, and is calculated using the partial derivatives of the simulated observation equation (equation 3.13) with respect to the elements of the state vector, evaluated using the nominal state vector $\bar{X}_n$. $T_{(n)}$ is computed by looping through the samples that make up a single observation instant or Gauss-Newton cycle (time instances $n-L$ to n where $L = 80$ as previously stated). During the iteration for any one sample $n-i$, the sensitivity matrix is first calculated using $\bar{X}_{n-i}$, then a block of rows of $T_{(n)}$ are calculated using the matrix multiplication $M(\bar{X}_{n-i})\Phi(-i.\tau)$. Recall from equation 3.18 that the sensitivity matrix dimensions depend on the length of the state vector and the number of observation variables. A combination of Doppler and bearing (split into sine and cosine) observations results in a total of 24 observation variables (3 observation variables on each of 8 receivers) per sample. A 10th degree polynomial in three dimensions is used for filtering, thus the state vector has a length of 33. The sensitivity matrix for a single sample therefore has dimensions of 24x33 (see section 3.3.2). Since CSXL library functions can only be called from Cn code when the matrices involved have dimensions that are multiples of 96, this matrix multiplication must be implemented manually.

There are 80 samples in a single Gauss-Newton algorithm cycle, and therefore 80 sensitivity matrices in the T matrix calculation. It was planned to calculate each of the sensitivity matrices on a processing element of the CSX600 processor. The two matrices $M(\bar{X}_{n-i})$ and $\Phi(-i.\tau)$ are shown in Figures 6.3 and 6.4 respectively. The areas shown as white space represent zeros.

	0	1	2 … 10	11	12	13 … 21	22	23	24 … 32
0	$D_x f_d$	$D_{xdot} f_d$	0 … 0	$D_y f_d$	$D_{ydot} f_d$	0 … 0	$D_z f_d$	$D_{zdot} f_d$	0 … 0
.			0 … 0			0 … 0			0 … 0
.			0 … 0			0 … 0			0 … 0
.			0 … 0			0 … 0			0 … 0
.			..			..			..
.			..			..			..
.			..			..			..
7			..			..			..
8	$D_x \cos\Psi$	0	..	$D_y \cos\Psi$	0	..	0	0	..
.		0	..		0	..	0	0	..
.		0	..		0	..	0	0	..
.		0	..		0	..	0	0	..
.		0	..		0	..	0	0	..
.		0	..		0	..	0	0	..
.		0	..		0	..	0	0	..
15		0	..		0	..	0	0	..
16	$D_x \sin\Psi$	0	..	$D_y \sin\Psi$	0	..	0	0	..
.		0	..		0	..	0	0	..
.		0	..		0	..	0	0	..
.		0	..		0	..	0	0	..
.		0	..		0	..	0	0	..
.		0	0 … 0		0	0 … 0	0	0	0 … 0
.		0	0 … 0		0	0 … 0	0	0	0 … 0
23		0	0 … 0		0	0 … 0	0	0	0 … 0

Figure 6.3: Structure of the observation sensitivity matrix showing sparse nature

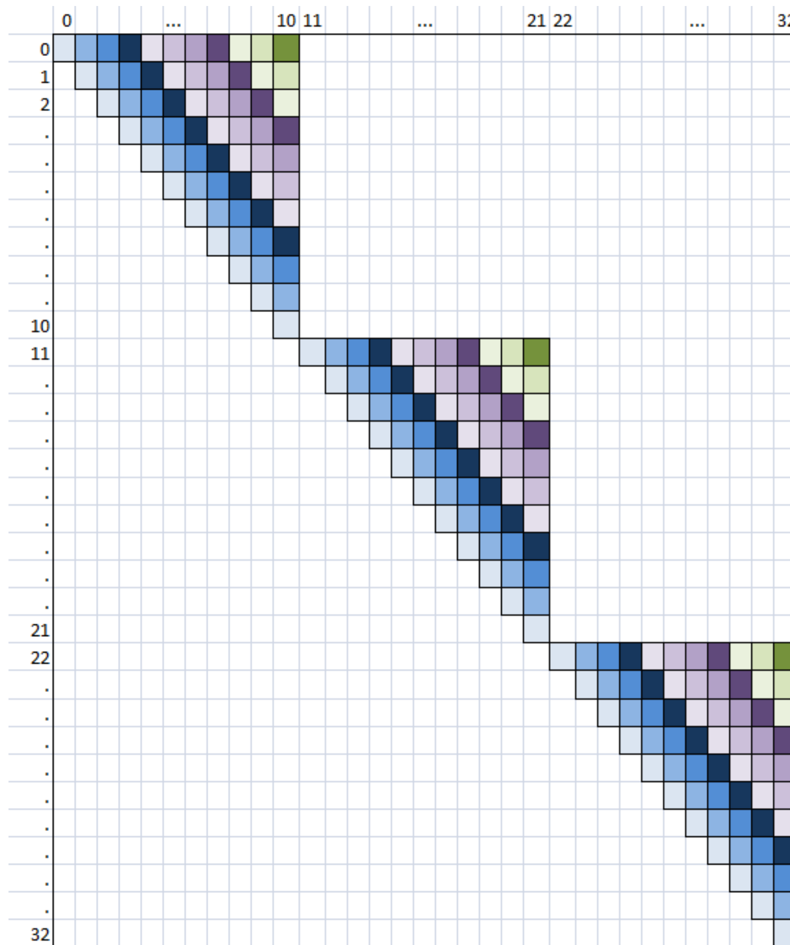

Figure 6.4: Graphical representation of state vector transition matrix structure showing sparse nature and data redundancy

$M(\bar{X}_{n-i})$ consists of the partial derivatives of each observation function for each of the eight receivers. The planned implementation calculated the partial derivatives for each receiver in a loop. Each iteration of the loop therefore calculated three rows of the sensitivity matrix, one for each observation variable. The transition matrix $\Phi(-i.\tau)$ consists of the same eleven values repeated over the sparsely populated matrix, and is dependent only on the sample instant and does not change between loop iterations (since it is receiver independent). The data requirement for the calculation on each processing element is shown in Table 6.1.

Table 6.1: Sensitivity matrix calculation poly data requirement per processing element

data structure	description	dimension	size
$\bar{X}_i$	nominal state vector	6	48B
X_{rcvr}	receiver co-ordinates	8x3	192B
$M(\bar{X}_{n-i})_k$	sensitivity values for one receiver	10	80B
$\Phi(-i.\tau)_{small}$	non redundant transition matrix	11	88B
T_{row}	one row of observation matrix	33x3	264B
*	intermediate variables	≈20	≈ 160B
	total space requirement		≈830B

The flow of control diagram describing the intended implementation on one processing element is shown in Figure 6.5. The data redundancy in the matrices involved mean that the calculation of each element in a row of the total observation matrix consists of just two double precision multiplications and an addition. This is immediately evident from the structure of the matrices in Figures 6.3 and 6.4. Lining up each of the rows of $M(\bar{X}_{n-i})$ with the columns of $\Phi(-i.\tau)$ shows that there are only ever a maximum of two non zero elements for the addition that make up each output matrix element. The calculation would normally consist of 33 multiplies and 32 additions, if the matrices involved were dense.

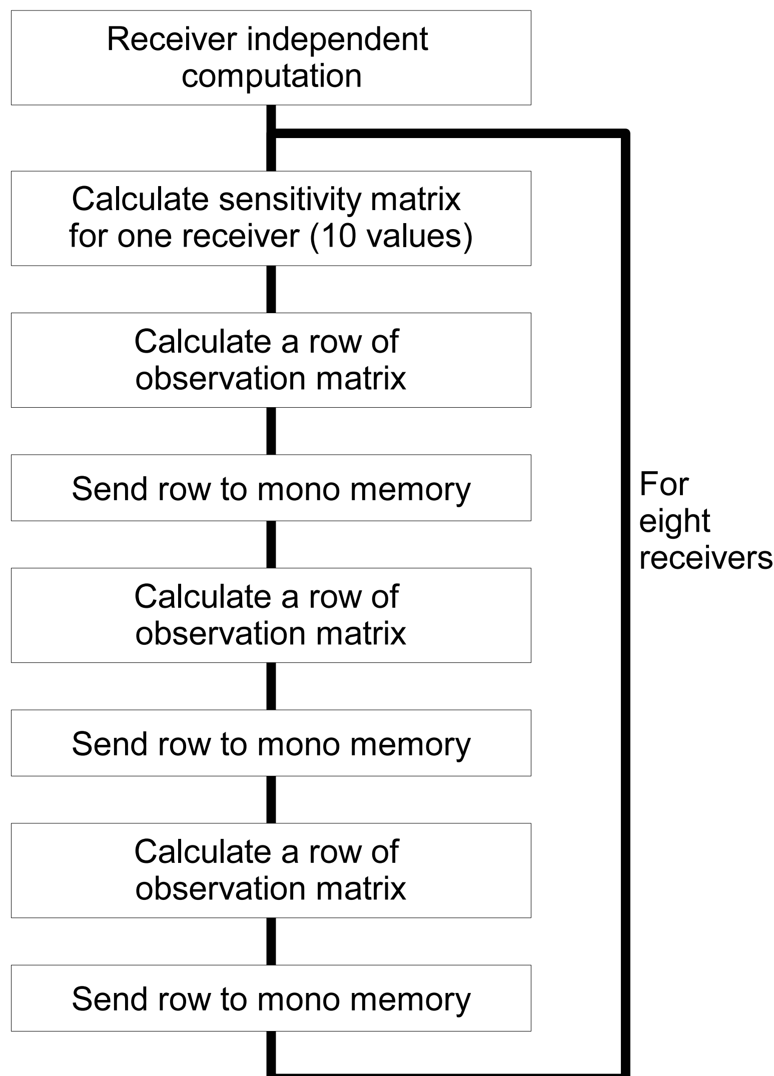

Figure 6.5: Total observation matrix computation on one ClearSpeed processing element

6.1.2 Prediction of performance of accelerated observation matrix calculation

The performance prediction calculation made the following assumptions:

- Pointer de-referencing and arithmetic takes an amount of time that is insignificant compared to double precision arithmetic and so can be disregarded. [15]

- The latency between the processing element poly SRAM and the registers is negligible compared to computation and the latencies between the other tiers of memory and may be disregarded [9].
- The bandwidth between the various tiers of memory time taken to set up transfers between the various tiers of memory is negligible
- The time taken per instruction were calculated from the cycle counts for various low level instructions from the ClearSpeed Instruction Set Reference Manual [15]. This document warns that the cycle count may vary according to external factors such as memory latency.
- The PCI-X 2.0 133MHz through-puts for the various data structures exchanged between host and coprocessor were determined empirically. Dummy data was repeatedly sent to and from the card and the profile data extracted by the runtime environment averaged to determine expected transfer rates. The expected transfer rates for the various data structures are shown in Table 6.2.
- The internal bandwidths of the ClearSpeed card were extracted from ClearSpeed's architecture overview [9].
- All the variables are stored as named variables in the poly memory space. This was not necessarily the case in the implementation. Certain intermediate values were defined in code as literal floating point values rather than named variables, which adversely affects performance.

Table 6.2: Transfer rates for total observation matrix calculation data structures

structure	dimensions	size (B)	source	destination	bandwidth	time (μs)
nominal state	80x3	1920	host	mono	44	43.6
Trows	33 (x3x8)	6336	poly	mono	30	211
T	1920x33	506880	mono	host	660	768

The number of instructions involved in the calculation of the total observation matrix for one Gauss-Newton iteration on the ClearSpeed card is shown in Table 6.3. The first column shows the various intermediate variables that are used in the calculation, and each row records the number of computations required for that variable.

Table 6.3: Number of instructions for T matrix calculation

# calcs	add	mult	div	comp	pow	sqrt	branch
Dopplerconst	0	1	2	0	0	0	0
A	2	3	0	0	0	0	0
B	2	0	0	1	3	1	0
receiver loop	8	0	0	0	0	0	9
x,y,z min	24	0	0	0	0	0	0
densquareds	8	0	0	8	16	0	0
densquared2	0	8	0	0	0	8	0
C	16	24	0	0	0	0	0
D	16	0	0	8	24	8	0
dfdx,y,z	72	48	96	0	48	0	0
dfdx,y,z dot	24	0	48	0	0	0	0
BigA	0	80	32	0	16	0	0
T 1st row loops	240	0	0	0	0	0	264
T 1st row calc	240	504	0	0	0	0	0
T 2nd row loops	264	0	0	0	0	0	288
T 2nd row	0	176	0	0	0	0	0
T 3rd row loops	264	0	0	0	0	0	288
T3rd row	0	176	0	0	0	0	0

The expected times for the calculation are shown in Table 6.4. The first row shows the expected time for each of the instructions involved as recorded in the ClearSpeed CSX600 instruction set reference manual [15]. The following rows are obtained by multiplying the times for each instruction by the number of times the instructions are used in each line of code.

Table 6.4: Predicted times for T matrix computation instructions

	add	mult	div	comp	pow	sqrt	branch
Time (μs)	2.38E-8	2.38E-8	6.86E-7	1.90E-8	3.62E-5	8.54E-6	9.52E-9
Dopplerconst	0.00E+0	2.38E-8	1.37E-6	0.00E+0	0.00E+0	0.00E+0	0.00E+0
A	4.76E-8	7.14E-8	0.00E+0	0.00E+0	0.00E+0	0.00E+0	0.00E+0
B	4.76E-8	0.00E+0	0.00E+0	1.90E-8	1.09E-4	8.54E-6	0.00E+0
receiver loop	1.90E-7	0.00E+0	0.00E+0	0.00E+0	0.00E+0	0.00E+0	8.57E-8
x,y,z min	5.71E-7	0.00E+0	0.00E+0	0.00E+0	0.00E+0	0.00E+0	0.00E+0
densquareds	1.90E-7	0.00E+0	0.00E+0	1.52E-7	5.79E-4	0.00E+0	0.00E+0
densquared2	0.00E+0	1.90E-7	0.00E+0	0.00E+0	0.00E+0	6.83E-5	0.00E+0
C	3.81E-7	5.71E-7	0.00E+0	0.00E+0	0.00E+0	0.00E+0	0.00E+0
D	3.81E-7	0.00E+0	0.00E+0	1.52E-7	8.69E-4	6.83E-5	0.00E+0
dfdx,y,z	1.71E-6	1.14E-6	6.58E-5	0.00E+0	1.74E-3	0.00E+0	0.00E+0
dfdx,y,z dot	5.71E-7	0.00E+0	3.29E-5	0.00E+0	0.00E+0	0.00E+0	0.00E+0
BigA	0.00E+0	1.90E-6	2.19E-5	0.00E+0	5.79E-4	0.00E+0	0.00E+0
T 1st row loops	5.71E-6	0.00E+0	0.00E+0	0.00E+0	0.00E+0	0.00E+0	2.51E-6
T 1st row calc	5.71E-6	1.20E-5	0.00E+0	0.00E+0	0.00E+0	0.00E+0	0.00E+0
T 2nd row loops	6.29E-6	0.00E+0	0.00E+0	0.00E+0	0.00E+0	0.00E+0	2.74E-6
T 2nd row	0.00E+0	4.19E-6	0.00E+0	0.00E+0	0.00E+0	0.00E+0	0.00E+0

Adding up the expected instruction execution and data transfer times gives a prediction of 5.27 ms for the development of the total observation matrix for one Gauss-Newton iteration. The observation perturbation vector calculation takes approximately 2.3 ms on average on the host. Therefore the flow of control shown in Figure 6.2, where the host calculates the observation perturbation vector and then waits for the total observation matrix to be computed, would be expected.

6.1.3 Prediction of performance of CSXL matrix multiplications

The matrix multiplications involved were found to not be well suited for ClearSpeed acceleration. Testing the speed of CSXL library calls against the host ATLAS libraries showed that all of the matrix multiplications involved were not a fit, as is shown in Figure 6.6. In most of the matrix sizes, fairly dramatic performance degradation was seen. This illustrates the main drawback to co-processor acceleration, that when the problem size or type does not fit the hardware of the co-processor, no significant performance benefit will be seen. In some cases, as in this one, performance degradation can even result. Consider the matrix multiplication: $C = A * B$ where A is an m by k matrix, B is a k by n matrix and C is the m by n matrix resulting from the multiplication of the two. ClearSpeed's DGEMM routine performs best when m and n are multiples of 192 and k is a multiple of 288 [11]. The only situation where any of these conditions are satisfied is the first multiply of the state vector perturbation vector (labeled *covMat x TTR-1* in Figure 6.6) where n=1920, however m=k=33 and the performance is still inferior to ATLAS.

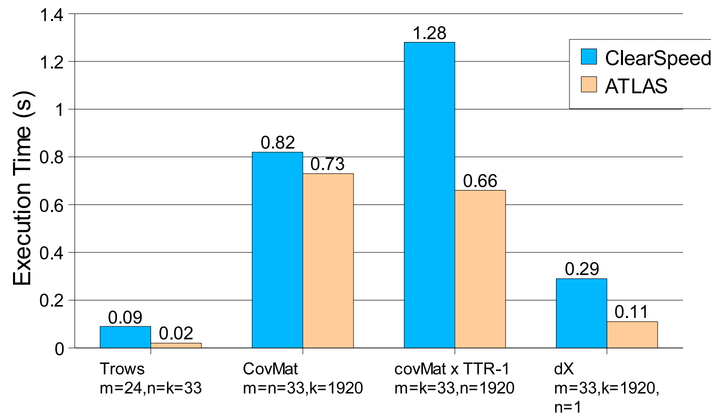

Figure 6.6: CSXL vs ATLAS matrix multiplication execution times. The main matrix multiplications of the algorithm were tested for performance, the graph shows execution times for 500 iterations of each multiplication. Here the symbols m, n and k represent the dimensions of the matrices in the multiplication where an m by k matrix has been multiplied by a k by n matrix, the resulting matrix being m by n. *Trows* represent the multiplication of the sensitivity matrix for a sample $M(X_i)$ (24 by 33), with the transition matrix for that sample Φ_i (33 by 33). *CovMat* represents the inverse covariance matrix calculation where $(T^T_{(n)}R^{-1}_{(n)})$ (33 by 1920) is multiplied by $T_{(n)}$ (1920 by 33). *covMat x TTR-1* is the multiplication of the covariance matrix, $(T^T_{(n)}R^{-1}_{(n)}T_{(n)})^{-1}$ (33 by 33), with $(T^T_{(n)}R^{-1}_{(n)})$ (33 by 1920). The perturbation vector dX is calculated by multiplying the 33 by 1920 result of the previous multiplication with the total observation perturbation vector $\delta Y_{(n)}$ (1920 by 1)

The matrix sizes shown in the graph correspond to the matrix multiplications show in Table 6.5. A and B are the matrices involved in the multiplication, where A is mxk, B is kxn and the output matrix is mxn. The times shown are in seconds and are for 100 calls of the matrix multiplication library function.

Table 6.5: Matrix sizes involved in Gauss-Newton filtering

A	B	m	n	k	output	CSXL (s)	ATLAS (s)
$M(\bar{X}_{(n)})$	Φ	24	33	33	24 rows of $T_{(n)}$	0.09	0.02
$T^T_{(n)}R^{-1}_{(n)}$	$T_{(n)}$	33	33	1920	$T^T_{(n)}R^{-1}_{(n)}T_{(n)}$	0.82	0.73
$(T^T_{(n)}R^{-1}_{(n)}T_{(n)})^{-1}$	$T^T_{(n)}R^{-1}_{(n)}$	33	1920	33	$(T^T_{(n)}R^{-1}_{(n)}T_{(n)})^{-1}T^T_{(n)}R^{-1}_{(n)}$	1.28	0.66
$(T^T_{(n)}R^{-1}_{(n)}T_{(n)})^{-1}T^T_{(n)}R^{-1}_{(n)}$	$\delta Y_{(n)}$	33	1	1920	$\delta Y_{(n)}$	0.29	0.11

6.2 ClearSpeed assisted Gauss-Newton algorithm

This section describes the final ClearSpeed assisted implementation, as well as the accuracy and performance profiling results.

6.2.1 Description of ClearSpeed assisted implementation

The implementation phase involved iterative coding and verification using black box testing on a functional unit by functional unit basis. This was done by sending relevant test data to the ClearSpeed card, running the unit in question on

the co-processor, then reading the output and testing for accuracy. If an error could not be identified by code examination after a failed unit test, the ClearSpeed debugger was used. This allowed tracing through the flow of control of the co-processor accelerated unit. Simultaneous tracing through the unassisted code allowed comparison of the sequential state of the host and accelerated versions in order to identify the erroneous section of code. The ClearSpeed implementation flowchart is shown in Figure 6.7. This shows how the processing is shared between the host and the accelerator board, as well as the interactions between them.

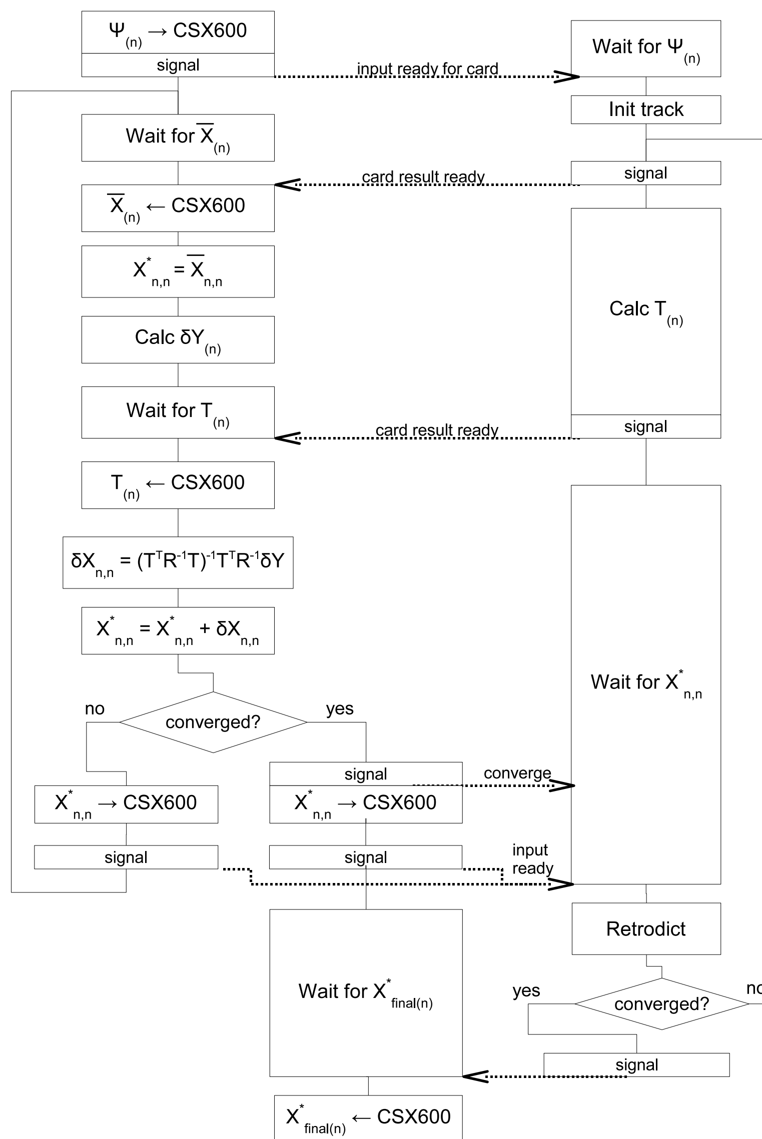

Figure 6.7: ClearSpeed accelerated Gauss Newton flow chart. Computations offloaded to the ClearSpeed card were track initialization, calculation of the total observation matrix and state vector retrodiction. Acceleration was attempted by removing redundant data and unrolling loops

During implementation of the algorithm it was decided to offload the track initialization and state vector retrodiction computations to the ClearSpeed card additionally to the total observation matrix calculation. The track initialization procedure contains unroll-able loops. The retrodiction calculation consists of matrix multiplication within an unroll-able loop. Since the matrices involved are sparse, processing was reduced by removing redundant data. The output of

retrodiction was previously the 10th degree polynomial state vector. Since only the 0th and 1st derivatives were required to replace the nominal state vector for consecutive iterations, (because the sensitivity matrix calculation requires only those two, see Appendix C), computation of higher order derivatives was removed from the retrodiction step.

The total observation matrix calculation was implemented as described in section 6.1.1.

6.2.2 Verification of ClearSpeed assisted implementation

The accuracy of the ClearSpeed accelerated code was tested in a similar fashion to the C implementation. A block diagram of the MCA is shown in Figure 6.8.

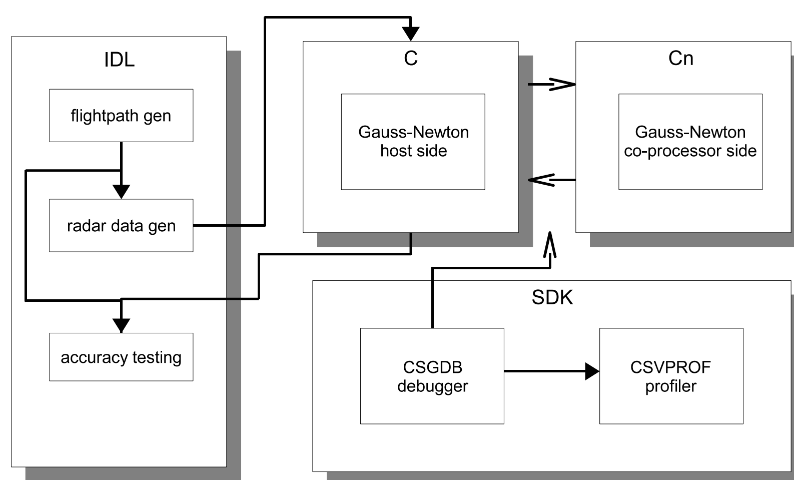

Figure 6.8: ClearSpeed accelerated Gauss-Newton MCA

The accuracy results verify the correct functionality of the ClearSpeed accelerated algorithm (Figure 6.9), however the results showed that the ClearSpeed accelerated implementation produced estimates an order of magnitude less accurate than the IDL version. This can be attributed to the differences in accuracy of the card side matrix multiplication and the IDL and BLAS library routines. This is supported by examining the average values of the perturbation vector ($\delta X_{(n)}$), which show that in the accelerated version perturbation vector was on average larger than that of the unaccelerated algorithm (Table 6.6)

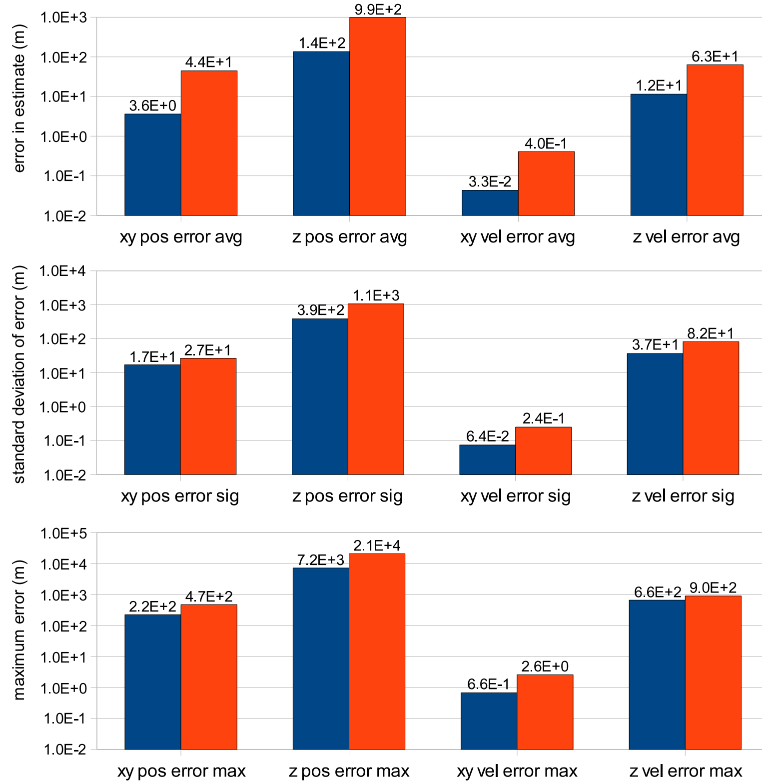

Figure 6.9: ClearSpeed accelerated accuracy results. The ClearSpeed results are an order of magnitude less accurate than those produced by the original IDL implementation, however still sufficiently to meet the stopping conditions of the algorithm as defined by Morrison [3]

Table 6.6: Comparative stopping rule statistics of C implementation and ClearSpeed assisted algorithm. The larger average perturbation vector of the accelerated implementation suggests that the ATLAS BLAS library implementation of matrix multiplication is more accurate than the custom written card side calculation.

	Iterations	average $\delta X_{(n)}$	Doppler residuals	bearings residuals
C	4.49162	5.46657	7.18154	0.00658
CSX	4.52189	5.65714	7.19699	0.00658

An example of the visual output of the IDL accuracy testing can be seen in Figure 6.10, which shows how the landing aircraft's X/Y co-ordinates are successfully tracked by the algorithm. As was the case with the IDL and C implementations the aircraft altitude cannot be successfully estimated due to there being insufficient available data from the limited number of receivers. This is discussed in chapter 7.

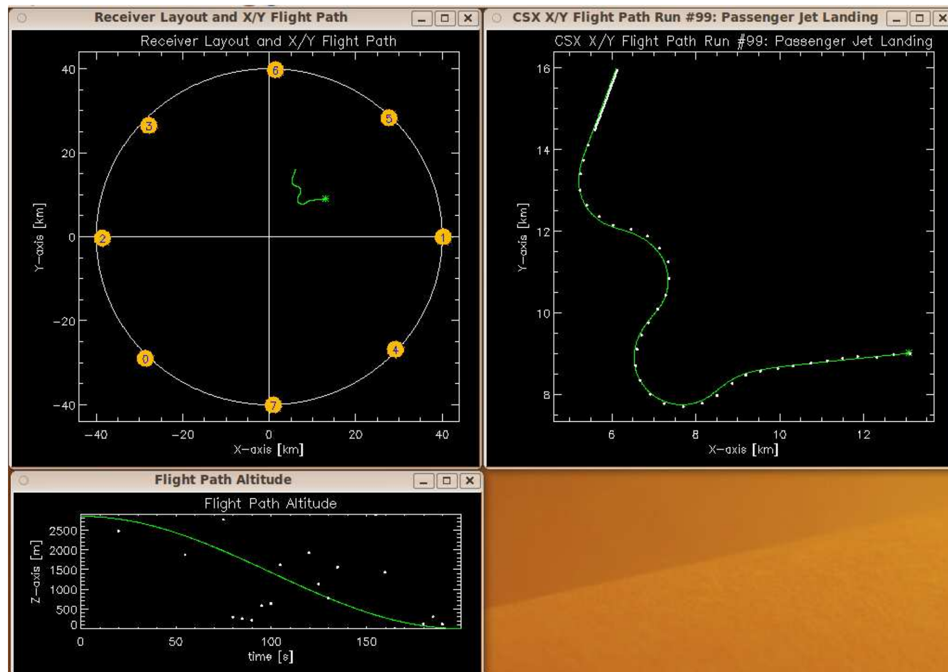

Figure 6.10: Accelerated tracking output visualization with IDL simulator program. The successful X/Y tracking of the aircraft is evident, as is the inaccuracy of the estimated altitude due to the limited number of receivers in the PCL radar network.

6.2.3 ClearSpeed accelerated algorithm profiling

The ClearSpeed accelerated algorithm was profiled using the ClearSpeed debugger. The output was viewed with the ClearSpeed visual profiler, and entered into a spreadsheet to allow its performance to be compared with the host version. The performance results are shown in the graphs of Figures 6.11 and 6.12, which show the profile of the host and ClearSpeed interactions and processing. The graphs are colour coded to show simultaneous host co-processor execution. For example the orange blocks in the figures shows that while the CSX600 processor is calculating the total observation matrix $T_{(n)}$, the host reads the nominal state vector $\bar{X}_{(n)}$ from the card, calculates the observation perturbation vector $\delta Y_{(n)}$ and then waits for the CSX600 calculation to complete. After this the red block shows that the card waits for the new state vector estimate $X^*_{n,n}$, while the card reads $T_{(n)}$, calculates $\delta X^*_{n,n}$ (and then $X^*_{n,n}$, which is not shown as the execution time is negligible) and then writes $X^*_{n,n}$ to the card.

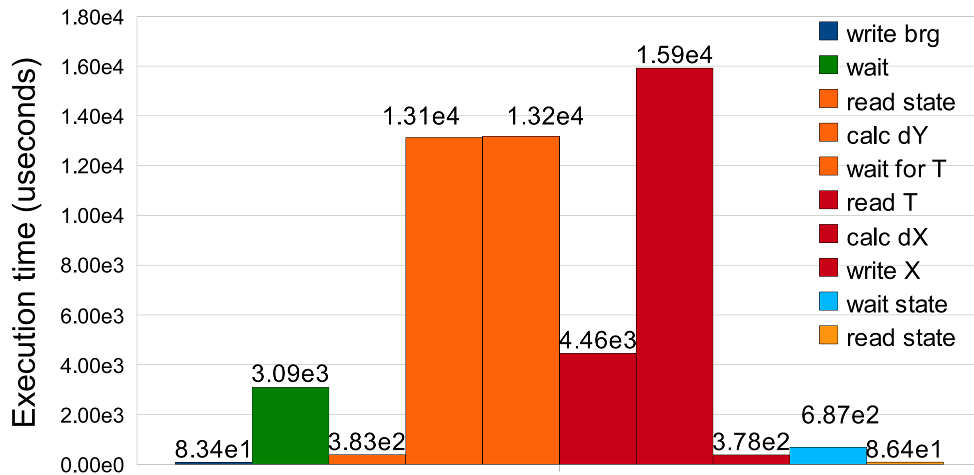

Figure 6.11: ClearSpeed accelerated host processing profile. Note that the first block (navy) is the write brg block that writes the bearing observations to the ClearSpeed card. The write X block writes the state vector (10th degree polynomial in 3 dimensions) to the card. The final read state block (yellow) reads the position and velocity retrodicted to all time instances in 3 dimensions back from the card.

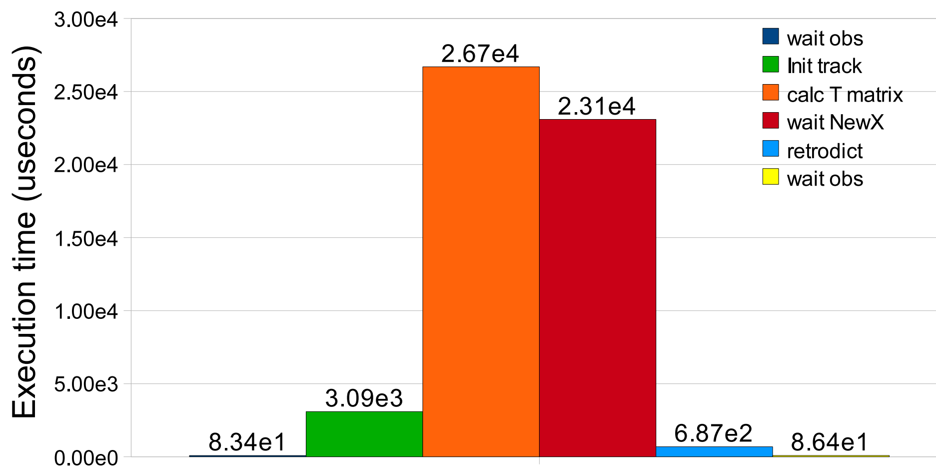

Figure 6.12: ClearSpeed accelerated card side processing profile. Note that the first and last block (navy and yellow respectfully) are the wait obs blocks in which the card waits for the bearings observations from the host. The first wait obs block happens while the bearings are being written to the card, and the last one occurs while the position and velocity estimate is being read back from the card.

The performance increase of the final accelerated implementation can be seen in Figure 6.13.

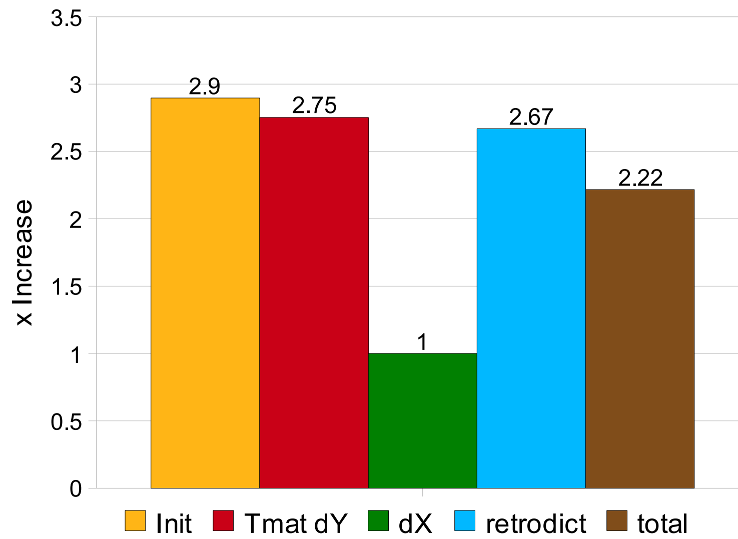

Figure 6.13: ClearSpeed accelerated performance increase graph

Conclusions

The accuracy results verify the correct functionality of the ClearSpeed accelerated algorithm, however the results showed that the ClearSpeed accelerated implementation produced estimates an order of magnitude less accurate than the IDL version. The final implementation resulted in about a 2.22 times speedup on average.

Chapter 7

Conclusions

7.1 Limitations of implementation

The implementation was capable of estimating target position and velocity in two dimensions. However target altitude estimations were not particularly accurate under the simulation conditions used. This was due to the limited number of receivers used in the simulated radar system. If more receivers were added to the PCL network, the input data set would rapidly multiply to an impractical size considering the limited available poly memory.

The accelerated Gauss-Newton implementation was tested for accuracy against the IDL version only. Therefore its functionality in the field is contingent on the real-world applicability of the pre-existing IDL code, which is yet to be tested. The IDL radar data generator used is fairly simplistic, therefore neither implementation has been tested with more realistic data. The accuracy results verified the correct functionality of the ClearSpeed accelerated algorithm, however the results showed that the ClearSpeed accelerated implementation produced estimates an order of magnitude less accurate than the IDL version. This can be attributed to the differences in accuracy of the card side matrix multiplication and the IDL and BLAS library routines, as supported by the difference in magnitudes of the average state vector perturbation vectors used in convergence testing (see Table 6.6).

7.2 Comments on programming with ClearSpeed

The ClearSpeed programming model has a two to three week learning curve, as it is fairly straight forward. Users experienced in parallel programming will find ClearSpeed programming even easier to adjust to. Due to the hefty price tag of the card, significant performance gains are required to make porting applications feasible. These performance gains will only be seen if the problem is either data parallel with high computation to data transfer ratio, or if linear algebra problems of a specific size are being solved. The amount of poly memory present is a major constraint, as is the bandwidth between mono and poly memory. These two constraints often result in performance penalties as the program is forced to continually swap data between mono and poly memory if the data set is too large. Single or even half precision should be used wherever possible.

7.3 Results of acceleration

Direct acceleration of the matrix operations via the provided ClearSpeed accelerated library functions was not feasible. The sizes of the matrices used were not conducive to ClearSpeed acceleration, even after tweaking the host assist environment variable (refer to section 2.3). Further investigation concluded that a moderate speedup could be attained through parallelization of the total observation matrix calculation. This acceleration was possible partly due to the sparse and data redundant nature of the input matrices. This allowed redundant data and calculations to be eliminated in the matrix multiplication. The CSXL library implements standard BLAS algorithms and therefore does not do any matrix value analysis and all matrices are assumed to be dense. The reasoning behind this is that the processing penalty in analyzing matrix values at runtime would outweigh the performance benefit that would result. Optimization at the programming stage can result in performance benefits but requires in depth knowledge and understanding of the algorithm and its data structures. The trade off is always between performance benefit and man hours spent tweaking the code.

The final implementation resulted in about a 2.22 times speedup on average. The $T_{(n)}$ matrix and $\delta Y_{(n)}$ vector calculation (executed in tandem on host and co-processor) had an approximate 2.75 times speedup on average. The design stage predicted an execution time of 5.14e3 microseconds for a single iteration of the $T_{(n)}$ and $\delta Y_{(n)}$ calculation, whereas the implementation averaged around 5.52e3 microseconds, a difference arising from the assumptions made in the performance prediction (see section 6.1.2).

Given the amount of programming effort required to achieve this moderate speedup, it can be concluded that the Gauss-Newton algorithm is not a good fit for ClearSpeed assisted acceleration.

7.4 Future work

Further work could be done in making the implementation more general, allowing for more receivers or different aircraft and radar parameters. Accounting for additional receivers would be particularly difficult as the data structures involved would increase in size significantly, however the second CSX600 processor on the board could perhaps be used to allow for this. Additionally it could be worth considering the implementation of a card-side matrix multiplication application fine tuned specifically for the matrix dimensions involved in the Gauss-Newton algorithm.

Given the expensive nature of the ClearSpeed cards, and the ultimate goal of reducing costs through PCL tracking, it would be worth considering cheaper alternative options to accelerate the Gauss-Newton filtering algorithms. This consideration is further motivated by the fairly moderate speedup achieved by the implementation, and the relative inaccuracy of the tracking estimates produced. These might include options such as GPGPU's and FPGA technologies, although the requirement for double precision might be restricting if FPGA's are to be used. The ClearSpeed accelerated code developed in this project might be used as a starting point for such an investigation or implementation.

For the Gauss-Newton tracking algorithm to ultimately be useful in the field, it should be able to track multiple targets simultaneously. While this appears to be an inherently parallel problem, the multiple target filtering algorithm still requires further work. Problems that are still to be solved centre on track identification (how to identify which data belongs to which target).

Bibliography

[1] K. Asanovic et al. The parallel computing laboratory at u.c. berkeley: A research agenda based on the berkeley view. Technical Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, Mar 2008.

[2] R. Lord and N. Morrison. Idl simulator program to implement dr n. morrison's doppler and bearing tracking, 2006.

[3] N. Morrison. *Gauss-Newton Smoothing for Radar and Tracking*. unpublished, 2009.

[4] N. Morrison. *Introduction to Sequential Smoothing and Prediction*. McGraw-Hill Book Company, 1969.

[5] H. Ranter. Airliner accident statistics 2006, statistical summary of world-wide fatal multi-engine airliner accidents in 2006. Technical report, Aviation safety network, January 2007.

[6] ClearSpeed Technology PLC. Technical Staff. *ClearSpeed Runtime Software User Guide*. ClearSpeed Technology PLC., September 2008.

[7] ClearSpeed Technology PLC. Technical Staff. *CSX600 Product Brief Datasheet*. ClearSpeed Technology PLC., 2006.

[8] ClearSpeed Technology PLC. Technical Staff. *Introduction to ClearSpeed Acceleration*. ClearSpeed Technology PLC., November 2007.

[9] ClearSpeed Technology PLC. Technical Staff. *Overview of Architecture: System Level (Host) Architecture and ClearSpeed Architecture*. ClearSpeed Technology PLC., November 2007.

[10] ClearSpeed Technology PLC. Technical Staff. *ClearSpeed Programming Model: Optimizing Performance*. ClearSpeed Technology PLC., November 2007.

[11] ClearSpeed Technology PLC. Technical Staff. *ClearSpeed CSXL User Guide*. ClearSpeed Technology PLC., April 2007.

[12] ClearSpeed Technology PLC. Technical Staff. *Introductory Programming Manual - The ClearSpeed Software Development Kit*. ClearSpeed Technology PLC., July 2007.

[13] ClearSpeed Technology PLC. Technical Staff. *ClearSpeed Advance X620 Accelerator Product Brief*. ClearSpeed Technology PLC., 2007.

[14] ClearSpeed Technology PLC. Technical Staff. *ClearSpeed CSXL User Guide*. ClearSpeed Technology PLC., August 2008.

[15] ClearSpeed Technology PLC. Technical Staff. *ClearSpeed CSX600/CSX700 Instruction Set Reference Manual*. ClearSpeed Technology PLC., August 2008.

Appendix A

This section describes the method of local linearization which is used to impose linearity onto the differential equation that governs the filter model used in Gauss-Newton tracking. In filter engineering the only way to estimate the state of a system that is modeled by a non-linear differential equation is to impose linearity, and if we fail to do so we cannot proceed [3].

First recall from equation 3.10 that $X_n = \bar{X}_n + \delta X_n$. Substituting this into a non-linear filter model results in equation 7.1 [3].

$$D\,(\bar{X}_n + \delta X_n) = F\,(\bar{X}_n + \delta X_n) \tag{7.1}$$

Expanding the left hand side of 7.1 gives equation 7.2. Equating the right hand side of equations 7.1 and 7.2 allows us to define the 1st derivative of the perturbation vector, as stated in equation 7.3.

$$D\,(\bar{X}_n + \delta X_n) = D\bar{X}_n + D\,\delta X_n = F(\bar{X}_n) + D\,\delta X_n \tag{7.2}$$

$$D\,\delta X_n = F\,(\bar{X}_n + \delta X_n) - F(\bar{X}_n) \tag{7.3}$$

For the sake of this explanation we will assume that the state vector X_n (and therefore the nominal vector $\bar{X}_n$ and the perturbation vector δX_n) consists of two elements. i.e. $X_n = (x_1, x_2)^T$. Note however that the following discussion applies just as well to the general case where X_n is a vector of arbitrary order. We can then rewrite the perturbation vector differential equation as:

$$D\begin{pmatrix} \delta x_1 \\ \delta x_2 \end{pmatrix} = \begin{pmatrix} f_1(\bar{x}_1 + \delta x_1, \bar{x}_2 + \delta x_2) \\ f_2(\bar{x}_1 + \delta x_1, \bar{x}_2 + \delta x_2) \end{pmatrix} - \begin{pmatrix} f_1(\bar{x}_1, \bar{x}_2) \\ f_2(\bar{x}_1, \bar{x}_2) \end{pmatrix} \tag{7.4}$$

As shown by Morrison [3] we can apply a Taylor's series expansion for two variables to each of the equations in 7.4 [3]. If we apply a Taylor's series expansion to the equation in the first row the result is equation 7.5 [3].

$$f_1(\bar{x}_1 + \delta x_1, \bar{x}_2 + \delta x_2) - f_1(\bar{x}_1, \bar{x}_2) = \delta x_1 D_{x_1} f_1(x_1, x_2)\,|_{\bar{X}_n} + \delta x_2 D_{x_2} f_1(x_1, x_2)\,|_{\bar{X}_n} \tag{7.5}$$

In 7.5 the following applies:

- D_{x_1} is the partial derivative with respect to x_1.
- D_{x_2} is the partial derivative with respect to x_2.
- $|_{\bar{X}_n}$ means that the partial derivatives are evaluated using the elements of $\bar{X}_n$.

Note that all the higher order terms have been dropped such that 7.5 is a 1st order expansion. This can be done because when functioning correctly the Gauss-Newton procedure iteratively increases the accuracy of our estimate $\bar{X}_n$ and therefore

δX_n eventually becomes negligible. Thus the higher order terms (which will contain powers of δx_1 and δx_2) should be insignificant [3]. If the same method of expansion is applied to the second equation and things are rearranged, equation 7.4 can be restated as 7.6 [3].

$$D\begin{pmatrix} \delta x_1(t) \\ \delta x_2(t) \end{pmatrix} = \begin{pmatrix} D_{x_1} f_1(x_1(t), x_2(t)) & D_{x_2} f_1(x_1(t), x_2(t)) \\ D_{x_1} f_2(x_1(t), x_2(t)) & D_{x_2} f_2(x_1(t), x_2(t)) \end{pmatrix}_{\bar{X}(t)} \begin{pmatrix} \delta x_1(t) \\ \delta x_2(t) \end{pmatrix} \tag{7.6}$$

We have generalized to any time instant t and in doing so stressed the time dependence of the state variables. Generalizing equation 7.6 to an arbitrarily sized state vector, we now have a linear form (equation 7.7) [3]. Thus we have gone from the non-linear equation describing the filter model to the linear form of the perturbation vector differential equation (assuming we have a nominal state vector $\bar{X}(t)$ that differs from the true state by the perturbation vector $\delta X(t)$).

$$D\,\delta X(t) = A(\bar{X}(t))\,\delta X(t) \tag{7.7}$$

In the preceding the matrix $A(\bar{X}(t))$ is known as the differential equation sensitivity matrix [3]. If X and F are m-vectors we define the sensitivity matrix in equation 7.8 [3].

$$[A(\bar{X}(t))]_{i,j} = \partial f_i(x_1 \ldots x_m)/\partial x_j \,|_{\bar{X}(t)} \quad i,j = 1 \ldots m \tag{7.8}$$

This states that the i, j^{th} element of the sensitivity matrix is calculated by taking the derivative of the ith function of F with respect to the jth element of X, evaluated using the elements of the $\bar{X}(t)$.

Appendix B

This appendix presents the partial derivatives of the observation equations with respect to the state vector variables. These equations are used to develop the total observation matrix which is the heart of the Gauss-Newton algorithm calculation.

Doppler equation partial derivatives

Recall the Doppler equation from 3.4:

$$f_d(x,y,z) = -{}^{2\pi}/_{\lambda} \left[\frac{(x-x_0)\dot{x} + (y-y_0)\dot{y} + (z-z_0)\dot{z}}{\left((x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2\right)^{1/2}} + \frac{(x-x_k)\dot{x} + (y-y_k)\dot{y} + (z-z_k)\dot{z}}{\left((x-x_k)^2 + (y-y_k)^2 + (z-z_k)^2\right)^{1/2}} \right] \tag{7.9}$$

The partial derivative of $f_d(x,y,z)$ with respect to x is:

$$D_x f_d(x,y,z) = -{}^{2\pi}/_{\lambda} \left[\frac{\dot{x}\left((x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2\right)^{1/2} - \frac{((x-x_0)\dot{x}+(y-y_0)\dot{y}+(z-z_0)\dot{z})(x-x_0)}{\left((x-x_0)^2+(y-y_0)^2+(z-z_0)^2\right)^{1/2}}}{(x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2} \right]$$

$$+ \left(-{}^{2\pi}/_{\lambda}\right) \left[\frac{\dot{x}\left((x-x_k)^2 + (y-y_k)^2 + (z-z_k)^2\right)^{1/2} - \frac{((x-x_k)\dot{x}+(y-y_k)\dot{y}+(z-z_k)\dot{z})(x-x_k)}{\left((x-x_k)^2+(y-y_k)^2+(z-z_k)^2\right)^{1/2}}}{(x-x_k)^2 + (y-y_k)^2 + (z-z_k)^2} \right] \tag{7.10}$$

The partial derivative of $f_d(x,y,z)$ with respect to y is:

$$D_y f_d(x,y,z) = -{}^{2\pi}/_{\lambda} \left[\frac{\dot{y}\left((x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2\right)^{1/2} - \frac{((x-x_0)\dot{x}+(y-y_0)\dot{y}+(z-z_0)\dot{z})y-y_0)}{\left((x-x_0)^2+(y-y_0)^2+(z-z_0)^2\right)^{1/2}}}{(x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2} \right]$$

$$+ \left(-{}^{2\pi}/_{\lambda}\right) \left[\frac{\dot{y}\left((x-x_k)^2 + (y-y_k)^2 + (z-z_k)^2\right)^{1/2} - \frac{((x-x_k)\dot{x}+(y-y_k)\dot{y}+(z-z_k)\dot{z})(y-y_k)}{\left((x-x_k)^2+(y-y_k)^2+(z-z_k)^2\right)^{1/2}}}{(x-x_k)^2 + (y-y_k)^2 + (z-z_k)^2} \right] \tag{7.11}$$

The partial derivative of $f_d(x,y,z)$ with respect to z is:

$$D_z f_d(x,y,z) = -{}^{2\pi}/_{\lambda} \left[\frac{\dot{z}\left((x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2\right)^{1/2} - \frac{((x-x_0)\dot{x}+(y-y_0)\dot{y}+(z-z_0)\dot{z})(z-z_0)}{\left((x-x_0)^2+(y-y_0)^2+(z-z_0)^2\right)^{1/2}}}{(x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2} \right]$$

$$+(-2\pi/\lambda)\left[\frac{\dot{z}\left((x-x_k)^2+(y-y_k)^2+(z-z_k)^2\right)^{1/2}-\frac{((x-x_k)\dot{x}+(y-y_k)\dot{y}+(z-z_k)\dot{z})(z-z_k)}{\left((x-x_k)^2+(y-y_k)^2+(z-z_k)^2\right)^{1/2}}}{(x-x_k)^2+(y-y_k)^2+(z-z_k)^2}\right] \tag{7.12}$$

The partial derivative of $f_d(x,y,z)$ with respect to $\dot{x}$ is:

$$D_{\dot{x}}f_d(x,y,z)=(-2\pi/\lambda)\left[\frac{(x-x_0)}{\left((x-x_0)^2+(y-y_0)^2+(z-z_0)^2\right)^{1/2}}+\frac{(x-x_k)}{\left((x-x_k)^2+(y-y_k)^2+(z-z_k)^2\right)^{1/2}}\right] \tag{7.13}$$

The partial derivative of $f_d(x,y,z)$ with respect to $\dot{y}$ is:

$$D_{\dot{y}}f_d(x,y,z)=(-2\pi/\lambda)\left[\frac{(y-y_0)}{\left((x-x_0)^2+(y-y_0)^2+(z-z_0)^2\right)^{1/2}}+\frac{(y-y_k)}{\left((x-x_k)^2+(y-y_k)^2+(z-z_k)^2\right)^{1/2}}\right] \tag{7.14}$$

The partial derivative of $f_d(x,y,z)$ with respect to $\dot{z}$ is:

$$D_{\dot{z}}f_d(x,y,z)=(-2\pi/\lambda)\left[\frac{(z-z_0)}{\left((x-x_0)^2+(y-y_0)^2+(z-z_0)^2\right)^{1/2}}+\frac{(z-z_k)}{\left((x-x_k)^2+(y-y_k)^2+(z-z_k)^2\right)^{1/2}}\right] \tag{7.15}$$

Cosine of bearing partial derivatives

Recall the cosine of bearing equation from 3.4:

$$\cos(\psi(x,y,z))=\frac{(x-x_k)}{\left((x-x_k)^2+(y-y_k)^2\right)^{1/2}} \tag{7.16}$$

The partial derivative of $\cos(\psi(x,y,z))$ with respect to x is:

$$D_x\cos(\psi(x,y,z))=\frac{\left((x-x_k)^2+(y-y_k)^2\right)^{1/2}-\frac{(x-x_k)^2}{\left((x-x_k)^2+(y-y_k)^2\right)^{1/2}}}{(x-x_k)^2+(y-y_k)^2} \tag{7.17}$$

The partial derivative of $\cos(\psi(x,y,z))$ with respect to y is:

$$D_y\cos(\psi(x,y,z))=\frac{-(x-x_k)(y-y_k)}{\left((x-x_k)^2+(y-y_k)^2\right)^{3/2}} \tag{7.18}$$

Sine of bearing partial derivatives

Recall the sine of bearing equation from 3.4:

$$\sin(\psi(x,y,z))=\frac{(y-y_k)}{\left((x-x_k)^2+(y-y_k)^2\right)^{1/2}} \tag{7.19}$$

The partial derivative of $\sin(\psi(x,y,z))$ with respect to x is:

$$D_x\sin(\psi(x,y,z))=\frac{-(x-x_k)(y-y_k)}{\left((x-x_k)^2+(y-y_k)^2\right)^{3/2}} \tag{7.20}$$

The partial derivative of $\sin(\psi(x,y,z))$ with respect to y is:

$$D_y \sin(\psi(x,y,z)) = \frac{\left((x-x_k)^2+(y-y_k)^2\right)^{1/2} - \frac{(y-y_k)^2}{\left((x-x_k)^2+(y-y_k)^2\right)^{1/2}}}{(x-x_k)^2+(y-y_k)^2} \tag{7.21}$$

Appendix C

This appendix describes the derivation of equation that results in the minimization of the sum of the weighted squared residuals $e(X^*_{n,n})$, in order to solve for the best state vector estimate $X^*_{n,n}$. This is done by differentiation of equation 3.31 with respect to $X^*_{n,n}$, setting the result to $\oslash$ and making $X^*_{n,n}$ the subject of the equation.

The differentiation is shown in equation 7.22 [3] where the symbol D_{X^*} signify vector differentiation w.r.t the elements of $X^*_{n,n}$.

$$\begin{aligned} & D_{X^*}\left((Y_{(n)} - T_{(n)}X^*_{n,n})^T R^{-1}_{(n)}(Y_{(n)} - T_{(n)}X^*_{n,n})\right) \\ &= \left(2D_{X^*}(Y_{(n)} - T_{(n)}X^*_{n,n})^T\right)\ R^{-1}_{(n)}(Y_{(n)} - T_{(n)}X^*_{n,n}) \\ &= -2T^T_{(n)}R^{-1}\,(Y_{(n)} - T_{(n)}X^*_{n,n}) = \oslash \end{aligned} \tag{7.22}$$

Multiplying out we then obtain equation 7.23:

$$T^T_{(n)}R^{-1}_{(n)}T_{(n)}X^*_{n,n} = T^T_{(n)}R^{-1}_{(n)}Y_{(n)} \tag{7.23}$$

Since $T^T_{(n)}R^{-1}_{(n)}T_{(n)}$ is non-singular we can solve for $X^*_{n,n}$ to produce equation 7.24 that minimizes the sum of weighted squared residuals and therefore is the best estimate of $X^*_{n,n}$ [3].

$$X^*_{n,n} = \left(T^T_{(n)}R^{-1}_{(n)}T_{(n)}\right)^{-1} T^T_{(n)}R^{-1}_{(n)}Y_{(n)} \tag{7.24}$$

Appendix D

This appendix presents the IDL, C and CSX implementation profiling data. Table 7.1 presents the IDL profiling data. The IDL simulator was run 10 times, each with a different flightpath as input. Each simulation consists of 37 observation instances, therefore times are averaged over 370 observation instances. The processing blocks shown are as described in section 4.3.

Table 7.1: IDL implementation profiling summary

Processing Block	Execution Time per update (uS)
Init	1.5835375E+3
dY	1.3881793E+4
partials	4.1192609E+4
T rows	3.3481308E+4
TtransRinv	9.6133592E+3
CovMat	1.8650126E+4
dX	3.2631370E+4
Xbar	4.3411125E+3
Latent time	2.3918734E+3
Total diffcorr time	1.5776709E+5

Table 7.2 presents the unassisted C implementation profiling data, and Table 7.3 profiling data of the ClearSpeed accelerated version. In both cases the information shown was averaged over 100 simulation runs, resulting in 3700 observation instances. The unassisted implementation was profiled using Intel's VTune Performance analyzer.

The provided debugger and visual profiler were used to test the performance of the ClearSpeed accelerated version. The information shown in Table 7.3 was averaged over 100 simulation runs, resulting in 3700 observation instances.

Table 7.2: Unassisted C implementation profiling summary. Note that total time denotes the time spent in a particular processing block including functions called from within that block, whereas self time denotes the amount of time spent excluding any functions called. *#_CALLS* represents the number of times the function is called during the 100 simulations. The term *leg* is used to mean observation instance or cycle. Therefore *PER LEG* is the time spent executing a processing block per observation instance (total time ÷ 3700). *PER ITERATION* denotes the time spent per iteration of the algorithm. *Iterations per leg* means iterations per observation instance or cycle. The processing blocks shown are as described in section 5.3

NAME	SELF_TIME (us)	TOTAL_TIME (us)	#_CALLS	PER LEG (us)	PER ITERATION (us)
main	6.3625E+6	6.7980E+8	1		
T Matrix	3.7651E+6	3.1726E+8	16255	85746.83	19517.89
Sensitivity	3.3087E+7	2.6363E+8	1300400	71250.27	16218.15
cblas_dgemm	1.1259E+6	1.0756E+8	3185980	29071.06	6617.22
T rows	3.4524E+6	4.9868E+7	1300400	13477.75	3067.84
initialise Track	1.0504E+7	3.4014E+7	3700	9192.99	9192.99
covariance matrix	2.0656E+4	3.0776E+7	16255	8317.77	1893.31
dX calculation	1.6033E+4	2.8127E+7	16255	7602.01	1730.39
retrodict	4.6548E+6	1.1368E+7	16255	3072.32	699.33
clapack_dgetrf	1.5270E+4	3.6106E+6	16255	975.85	222.12
clapack_dgetri	1.5285E+4	1.9772E+6	16255	534.37	121.64

Simulations	100
Legs	3700
Legs per simulation	37
Iterations	16255
Average Iterations per leg	4.4

Table 7.3: ClearSpeed accelerated implementation profiling summary. The top row labels are for the ClearSpeed card side execution blocks, the second row shows the host execution blocks. The table shows overlapping host and ClearSpeed execution.

CSX	wait	Init track	calculate T matrix			wait for NewX			retrodict	wait obs	TOTAL
host	write brg	wait	read state	calc dY	wait for T	read T	calc dX	write X	wait state	read state	CYCLE
Cycle	Time	Time	Time	Time	Time	Time	Time	Time	Time	Time	Time
1	150181	3084	669	18662	18716	17014	33080	542	878	97	242923
2	82	3078	326	9230	9456	2929	15907	262	433	83	41786
3	92	3071	242	6899	7123	2185	12077	197	318	87	32291
4	82	3071	242	6900	7128	2192	12252	198	316	86	32467
5	80	3071	238	6978	7226	2193	11905	204	327	84	32306
6	81	3072	406	11503	11856	3649	19867	327	540	85	51386
7	83	3073	239	6908	7105	2242	12175	202	325	83	32435
8	83	3071	996	27882	28061	8768	48096	795	1485	83	119320
9	83	3552	810	9212	9001	2940	16175	274	424	87	42558
10	82	3071	236	6904	7122	2397	11959	199	328	86	32384
11	84	3070	246	6903	7110	2191	12129	198	857	97	32885
12	86	3073	234	6928	7089	2188	12168	200	326	83	32375
13	81	3077	495	14012	14022	4381	23889	407	639	83	61086
14	81	3079	239	6911	7111	2196	12151	206	321	83	32378
15	82	3080	704	16450	13895	5191	53447	470	2983	108	96410
16	86	3052	308	6996	7009	2269	12009	197	226	114	32266
17	95	3101	356	9263	9393	2921	15874	263	430	86	41782
18	83	3086	730	21008	21025	6582	35766	605	961	84	89930
19	87	3082	484	14063	13969	4445	23891	403	648	83	61155
20	80	3084	646	18658	18713	5845	31810	539	845	85	80305
21	89	3084	383	111933	112148	35533	18258	3217	5212	86	289943
22	81	3082	248	6932	7091	2195	11984	199	320	84	32216
23	81	3081	493	14101	13927	4369	23808	395	647	83	60985
24	81	3079	163	4622	4729	1553	8018	132	212	84	22673
25	80	3078	239	6974	7040	2187	12831	204	326	86	33045
26	89	3078	487	13890	14121	4388	24308	411	652	83	61507
27	82	3077	236	6974	7053	2183	11939	198	318	83	32143
28	81	3079	239	6996	7031	2186	11971	199	319	84	32185
29	82	3076	240	6955	7072	2184	12204	200	319	90	32422
30	81	3076	246	6948	7073	2186	12004	197	320	84	32215
31	83	3080	236	6940	7072	2183	11956	200	325	85	32160
32	88	3078	319	9258	9430	2924	15959	279	428	86	41849
33	83	3077	566	16361	16336	5116	27784	461	757	84	70625
34	81	3081	241	6948	7072	2194	12044	206	325	84	32276
35	80	3077	241	6939	7080	2188	12213	206	310	86	32420
36	81	3073	405	11560	11787	3643	20222	331	550	85	51737
37	85	3073	329	9230	9454	2989	16244	263	469	71	42207
AVG	83.36	3089.92	382.61	13130.57	13179.62	4457.27	18258.17	378	687	86.35	58190.16

Appendix E

This appendix presents the C and CSX implementation accuracy data. Table 7.4shows a summary of the IDL accuracy results. Table 7.5 the unassisted C version, and Table 7.6 the ClearSpeed accelerated version. The results were averaged over 60 simulation results, each consisting of 37 observation instances. Thus the data is based on 2200 observation instances or Gauss-Newton cycles. Each observation instant consists of 80 samples of the state vector, which is reduced from position and its derivatives up to the 10th order in 3 dimensions which are developed by the algorithm, down to just position and velocity in 3 dimensions.

Table 7.4: IDL accuracy result summary

x/y position error average	z position error average	x/y velocity error average	z velocity error average
3.62175e0	1.35954e2	4.28028e-2	1.15105e1
x/y position error sigma	z position error sigma	x/y velocity error sigma	z velocity error sigma
1.71614e1	3.85369e2	7.44381e-2	3.67859e1
x/y position error maximum	z position error maximum	x/y velocity error maximum	z velocity error maximum
2.23577e2	7.20497e3	6.73337e-1	6.60119e2

Table 7.5: Unassisted C accuracy result summary

x/y position error average	z position error average	x/y velocity error average	z velocity error average
1.79546e0	1.21235e2	4.38044e-2	1.07059e1
x/y position error sigma	z position error sigma	x/y velocity error sigma	z velocity error sigma
1.14358e1	3.46901e2	8.29942e-2	3.25287e1
x/y position error maximum	z position error maximum	x/y velocity error maximum	z velocity error maximum
2.64009e2	6.63206e3	1.09669e0	5.12389e2

Table 7.6: ClearSpeed accelerated accuracy result summary

x/y position error average	z position error average	x/y velocity error average	z velocity error average
4.43928e1	9.88423e2	4.08008e-1	6.33252e1
x/y position error sigma	z position error sigma	x/y velocity error sigma	z velocity error sigma
2.65135e1	1.06132e3	2.50293e-1	8.20934e1
x/y position error maximum	z position error maximum	x/y velocity error maximum	z velocity error maximum
4.73207e2	2.07640e4	2.56812e0	8399293e2