Accelerating Gauss-Newton Filters on FPGAs

Jean-Paul Costa da Conceicao

A dissertation submitted to the Department of Electrical Engineering,
University of Cape Town, in fulfilment of the requirements
for the degree of Master of Science in Engineering.

Cape Town, December 2010

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.

.....

Jean-Paul da Conceicao

Cape Town

17th December 2010

Abstract

Radar tracking filters are generally computationally expensive, involving the manipulation of large matrices and deeply nested loops. In addition, they must generally work in real-time to be of any use. The now-common Kalman Filter was developed in the 1960's specifically for the purposes of lowering its computational burden, so that it could be implemented using the limited computational resources of the time. However, with the exponential increases in computing power since then, it is now possible to reconsider more heavy-weight, robust algorithms such as the original non-recursive Gauss-Newton filter on which the Kalman filter is based[54]. This dissertation investigates the acceleration of such a filter using FPGA technology, making use of custom, reduced-precision number formats.

Acknowledgements

There are a number of people and organisations I would like to thank, and without whom this dissertation would not be possible:

Firstly I would like to thank Professor Michael Inggs for his supervision, and for arranging the funding for my postgraduate studies. Thanks also to the SANDF for providing the funding for my second year of study.

The Centre for High Performance Computing (CHPC) in Rosebank, for the use of their facilities, friendly staff, and for the numerous courses and conferences I was able to attend and benefit from while working there. I would especially like to thank Dr Jeff Chen and Dr Happy Sithole for welcoming me and treating me as part of the organisation from day one, even though I was not on a CSIR studentship.

Everyone at the Advanced Computer Engineering (ACE) lab at the CHPC, who I had the pleasure of working with: Nick Thorne, Jane Hewitson, Dave MacLeod, Andrew Woods, Jason Salkinder, Andrew van der Byl, and Ray Hsieh. For all the help and support over the past two years, and for the after-hours beers, hikes and braais we enjoyed together. I have learnt a lot from them all, and this dissertation really would not have been possible without them.

Dr Norman Morrison of UCT, for the many hours spent teaching me the basics of filter theory, and for making the latest drafts of his soon-to-be published textbook on the subject available to me. I appreciate his time and effort, and his personal tuition really helped me to grasp the concepts a lot quicker than I would otherwise have been able to. I would also like to thank Dr Richard Lord for his helpful email correspondence.

Joe Milburn, for always being happy to help me understand and build on his work, from the smallest technical difficulties to providing insight and opinions on the various problems I had. His input was a great help to me.

Finally I would like to thank my family for all their help and support during my studies.

Contents

Declaration i

Abstract ii

Acknowledgements iii

List of Symbols xi

Nomenclature xii

1 Introduction 1

1.1 Problem Statement 1

1.1.1 The Computing Brick Wall 2

1.1.2 Reconfigurable Computing 4

1.2 Algorithm of Study 5

1.3 Objectives 6

1.4 Scope 7

1.5 Related Work 7

1.5.1 Previous study 7

1.5.2 Accelerating Radar Algorithms 8

1.5.3 HPC with FPGAs 8

1.5.4 Floating-Point on FPGAs 9

1.6 Plan of Development 10

2 Background 11

2.1 Data Smoothing 11

2.2 The Gauss-Newton Tracking Algorithm 11

2.2.1 Simple Example 12

2.2.2 Our Case . . . 14
2.3 Hardware Platform of Study . . . 17
2.3.1 Xilinx Virtex Family . . . 17
2.4 Number systems . . . 18
2.4.1 Fixed-Point Numbers . . . 18
2.4.2 Floating-Point Numbers . . . 18
2.4.3 Computing Just Right . . . 20
2.5 Experimental Setup . . . 20
2.5.1 Dataset . . . 21

3 Algorithm analysis **24**
3.1 Code Profiling . . . 24
3.1.1 Finding The Bottleneck . . . 24
3.1.2 Initial Bottleneck Analysis . . . 28
3.2 Arithmetic Arrangement and IPL . . . 32
3.3 Number Precision Investigation . . . 34
3.3.1 Floating-Point . . . 35
3.3.2 Fixed-Point . . . 37
3.4 Conclusions . . . 43

4 System Design **45**
4.1 Introduction . . . 45
4.1.1 Parallelism . . . 46
4.1.2 Pipelining . . . 47
4.1.3 Shifting . . . 48
4.1.4 Number Format Conversion . . . 50
4.1.5 Use of DSP Blocks . . . 50
4.2 Floating-Point design . . . 51
4.2.1 Xilinx Floating-Point Cores . . . 51
4.3 Fixed-Point Design . . . 52
4.4 Investigation Using Bishop Libraries . . . 56
4.5 Conclusions . . . 58

5 Implementation **59**
5.1 Implementation . . . 59
5.1.1 Coding . . . 59
5.1.2 Simulation . . . 60
5.1.3 Synthesis . . . 60
5.2 Verification . . . 61
5.2.1 Implementation Experiment on Coprocessor Card . . . 61
5.3 Testing . . . 64
5.4 Conclusions . . . 66

6 Results **67**
6.1 Accuracy Results . . . 67
6.2 FPGA Hardware . . . 67
6.2.1 Resource Usage . . . 67
6.2.2 Achievable Clock Speed . . . 72
6.2.3 Latency . . . 74
6.2.4 Power . . . 74
6.3 Projected Speedup . . . 74
6.3.1 Scaling . . . 75

7 Conclusions **77**
7.1 Comments on FPGA Development . . . 78
7.2 Future Work . . . 78

A Precision Profiling Functions **80**
A.1 Floating Point . . . 80
A.2 Fixed Point . . . 82

B Detailed Fixed-Point Precision Profiling Results **85**

C Python Scripts **96**
C.1 Reduced-Precision Simulation . . . 96
C.2 FPGA Resource Usage Trials . . . 101
C.3 Verification Scripts . . . 111

D Square-Root Shifting **115**

E Detailed Error Comparisons **117**

Bibliography **123**

List of Figures

1.1 Trends in the computing industry . . . 3
1.2 FPGA vs Processor efficiency . . . 5

2.1 Gauss-Newton Algorithm . . . 16
2.2 Floating-Point Number Format . . . 19
2.3 Block diagrams showing the workings of the existing code. . . . 22
2.4 Examples of Radar Target Tracks . . . 23

3.1 Performance Profile of the Algorithm . . . 26
3.2 Pseudocode and visualisation of the performance bottleneck . . . 27
3.3 Accuracy vs Precision for Reduced Precision Floating-Point . . . 36
3.4 Average Iterations to Convergence for Reduced Precision Floating-Point . . . 37
3.5 AccelChip Survey Results . . . 38
3.6 Accuracy vs Precision and Dynamic Range for Fixed-Point . . . 40
3.7 Fixed-Point Data Flow Model . . . 42
3.8 Resource Requirements of Flexible vs Fixed Word Widths for Fixed-Point . . . 44

4.1 Block Diagram of Hardware Design . . . 46
4.2 Visualisation of Exploited Parallelism . . . 47
4.3 Gantt Chart Showing Critical Path of Design . . . 48

5.1 Hardware Verification Process . . . 62
5.2 DIMETalk Network Used with Nallatech Coprocessor Card . . . 64
5.3 Hardware Testing Process . . . 65

6.1 Accuracy Results . . . 68
6.2 Distribution of FPGA Resources in Designs . . . 70
6.3 Scaling of Resource Requirements with Processing Elements . . . 73

6.4 Projected Speedup 76

B.1 X/Y Coordinate Accuracy for Varying Integer Lengths 86
B.2 Z Coordinate Accuracy for Varying Integer Lengths 87
B.3 Velocity X/Y Coordinate Accuracy for Varying Integer Lengths 88
B.4 Velocity Z Coordinate Accuracy for Varying Integer Lengths 89
B.5 Average Iterations to Convergence for Increasing Integer Width 90
B.6 X/Y Coordinate Accuracy for Varying Fraction Lengths 91
B.7 Z Coordinate Accuracy for Varying Fraction Lengths 92
B.8 Velocity X/Y Coordinate Accuracy for Varying Fraction Lengths 93
B.9 Velocity Z Coordinate Accuracy for Varying Fraction Lengths 94
B.10 Average Iterations to Convergence vs Fraction Length 95

E.1 Detailed X/Y Position Accuracies 118
E.2 Detailed Z Position Accuracies 119
E.3 Detailed X/Y Velocity Accuracies 120
E.4 Detailed Z Velocity Accuracies 121
E.5 Detailed Average Iterations to Convergence Results 122

List of Tables

2.1 FPGA Price Comparison 18

3.1 Input Variable Ranges 41
3.2 Fixed-Point Sizing Rules 42

4.1 FIFO/RAM Shifter Comparison 49
4.2 Divider Core Comparison 56
4.3 VHDL Library Design Summary 57

5.1 Nallatech Coprocessor Card Specifications 63

6.1 Resource Usage Results: Synthesis 69
6.2 Resource Usage Results: Post Place-And-Route 71

List of Symbols

M — Sensitivity matrix
T — Total Observation Matrix
X — State Vector
Y — Observation Vector
Φ — Transition Matrix
f — Doppler
ψ — Azimuth

Nomenclature

ADC	:	Analogue-to-Digital Converter
ASIC	:	Application Specific Integrated Circuit
Azimuth	:	Angle in a horizontal plane, relative to a fixed reference, usually north or the longitudinal reference axis of the aircraft or satellite.
BRAM	:	Block-RAM
CORDIC	:	COordinate Rotational DIgital Computer, an algorithm for efficient hardware implementation of trigonometry.
Device	:	Peripheral controlled by the Host in a heterogeneous computing system.
Doppler frequency	:	A shift in the radio frequency of the return from a target or other object as a result of the object's radial motion relative to the radar.
DP	:	Double Precision. 64-bit floating point number format with a sign bit, 11 exponent bits, and 52 mantissa bits.
DMA	:	Direct Memory Access
DSP	:	Digital Signal Processing/Processor
FIFO	:	First-In, First-Out. A method of organising data in memory.
FPGA	:	Field-Programmable Gate Array
FPU	:	Floating-Point Unit
GNA	:	Gauss-Newton Algorithm
GPU	:	Graphics Processing Unit
GUI	:	Graphical User Interface
Host	:	Computer which controls and interfaces with the Device in a heterogeneous computing system.
HPC	:	High Performance Computing
ILP	:	Instruction Level Parallelism
LUT	:	LookUp Table. Memory structure which is the fundamental building block of FPGA fabric.
MCA	:	Master Control Algorithm
NaN	:	Not a Number

PCL	:	Passive Coherent Location
PE	:	Processing Element
Range	:	The radial distance from a radar to a target.
RCS	:	Radar Cross Section. A measure of the reflectivity of a radar target.
SAR	:	Synthetic Aperture Array, a technique for achieving increased resolution for radar systems.
SIMD	:	Single Instruction, Multiple Data
SOC	:	System On Chip
SP	:	Single Precision. 32-bit floating point number format with a sign bit, 8 exponent bits, and 23 mantissa bits.

Chapter 1

Introduction

1.1 Problem Statement

Most remote-sensing applications which observe and track a moving target require some form of data smoothing, to reduce the effect of measurement noise and to clarify the result. Modern radar data-smoothing algorithms (otherwise known as "state estimators" or simply "filters") are based on various forms of the minimum variance algorithm developed in the mid-1930's. This algorithm in its original form is computationally expensive, involving the multiplication of large matrices, large matrix inversions, and nested loops. An additional obstacle to its practical application is that it is generally needed to process data in real-time to be of any use, for obvious reasons in applications such as defence and air-traffic control, and also because the storage space requirements for raw data can quickly become impractically large.

It was because of these difficulties that the now-common Kalman Filter was developed in the 1960's - only a more computationally light-weight solution could be implemented using the limited resources of the time. The Kalman filter uses a recursive form of the minimum-variance algorithm, and was originally developed specifically for the purpose of satellite-tracking. It has since become the standard data smoothing algorithm used in industry, and consideration of the original non-recursive form of the minimum variance algorithm for radar engineering applications is now rare. However, with the exponential increases in computing power over the last few decades, and the additional performance boosts to be gained from High-Performance-Computing (HPC) methods, it is now possible to reconsider more heavy-weight, robust algorithms, which can offer significant advantages over the Kalman Filter when properly used[54].

This dissertation investigates the acceleration of such a filter using reconfigurable computing technology. The algorithm is a Gauss-Newton Polynomial Filter developed by Dr Norman Morrison, Dr Richard Lord and Professor Michael Inggs of the Radar Remote Sensing Group at the University of Cape Town. It is designed for a Passive Coherent Location (PCL) radar system, and takes as input Doppler and bearing readings from multiple receivers. The targeted hardware platforms for the de-

sign are Xilinx Virtex-4 and Virtex-5 series FPGAs, working in tandem with an Intel Xeon 4-core processor.

1.1.1 The Computing Brick Wall

There is another reason, besides the needs of filter algorithms, for investigating alternative processing methods: they are fast becoming necessary for computing in general.

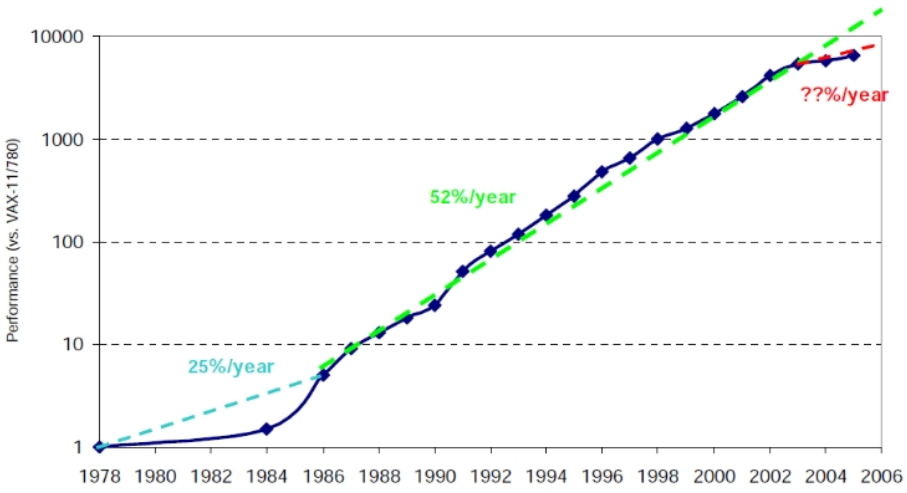
Figure 1.1a shows a graph of the increases in computational power over time between 1978 and 2006. From the graph we observe that although microprocessors have improved at a steady rate of about 52% per year since 1986, the rate has dropped since 2002. Moore's Law has however remained true (see figure 1.1b), and the drop in performance is not due to decreasing transistor density, but instead three 'walls' which have been encountered in the industry[44]:

- **Power Wall:** the linear relationship between clock frequency and power consumption in processors has meant that power supply is becoming a major limiting factor of processor designs: if trends continue, server farms will become prohibitively power-hungry and expensive, mobile devices will have unacceptably short battery-lives, and keeping CPUs sufficiently cool will become impractical or impossible. We thus cannot expect the same rate of improvement by simply increasing processor clock speeds.

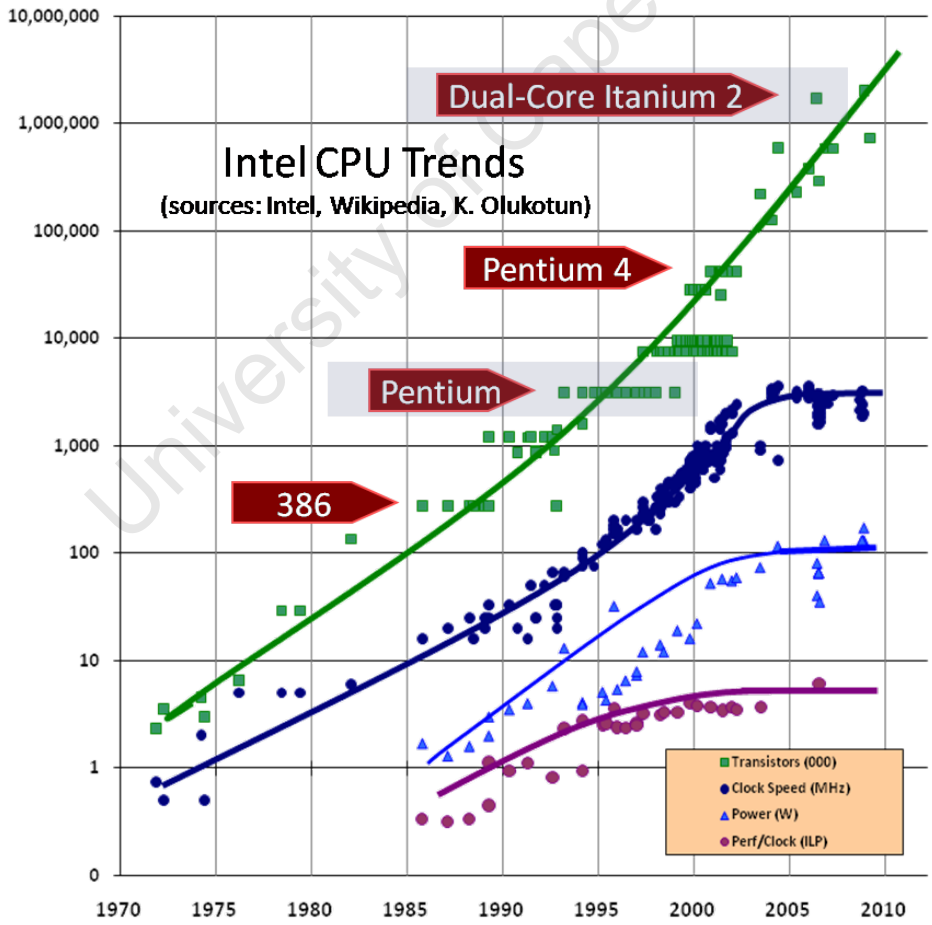
- **Memory Wall:** memory technologies have always advanced at a much slower rate than processing technologies; though they have also improved exponentially, their exponent is much smaller than that of processors, and the difference between exponents thus grows exponentially itself[50, 49]. As a result we are in a situation where memory loads and stores are orders of magnitude slower than calculations, and most of the area on a microprocessor's silicon die has to be set aside for cache systems which are steadily growing larger and more complex. This problem will only continue to worsen unless novel computing architectures are considered.

- **ILP Wall:** instruction-level-parallelism is exploited when compilers and processors identify operations which can be performed independently, and then perform these independent operations simultaneously. Techniques such as pipelining, out-of-order execution, and branch prediction are all methods used to achieve this in modern processors. Unfortunately, there are diminishing returns in implementing more and more ILP [41], mainly because it results in hardware which is more suited to a specific problem and is less useful for general-purpose computing.

According to [44] and many other researchers, *"Power Wall + Memory Wall + ILP Wall = Brick Wall"* and the only way forward, if we are to maintain the exponential growth rate we have enjoyed since the birth of computers, is to consider novel architectures that make use of some form of parallelism, such as multi-core processors or heterogeneous computing systems. To quote [66], "the free



(a) A graph of processor performance from 1978 to 2006 using integer benchmarks[41].



(b) Graph contrasting the recent trends of transistor count with those of clock speed, power and ILP in Intel processors[66]. Although clock speed and power hit a wall between 2000 and 2005, Moore's law has continued as expected.

Figure 1.1:

lunch is over", and programmers can no longer have their code accelerated by simply running it on the next generation of processors with faster clock speeds.

1.1.2 Reconfigurable Computing

One option for an HPC platform is reconfigurable computing - the most successful example of which is the Field Programmable Gate Array (FPGA). FPGAs were originally designed for the relatively simple task of implementing glue-logic between other components, or fast hardware prototyping. A rapid increase in reconfigurable logic densities, on-board RAM sizes, and dedicated hardware components has meant that these devices are now useful for more demanding tasks, such as Digital Signal Processing (DSP), and HPC in general. Reconfigurable computing technology has been used successfully for accelerated computing for over a decade[25, 45], and though at first they were confined to a relatively limited subset of computing tasks, recently they have been shown to be well suited to a range of problems, including those based on floating-point operations[67, 35].

Trends since 1997 have shown that FPGA computational densities are increasing faster than conventional microprocessors, as illustrated in figure 1.2 below. To quote[23]: "It clearly shows the inability of microprocessors to efficiently turn increasing die area and speed into useful computation." The reason for these order-of-magnitude differences is the simple structure of FPGAs, which scales better than processor architectures. It must be noted however that with this simple hardware structure comes the penalty of increased development effort when using FPGAs.

FPGA architectures are also well suited to use in a radar applications; they are very power-efficient thanks to their low clock speeds (making them orders of magnitude less power hungry than conventional microprocessors and GPUs), are easy to upgrade remotely, with no need for physical hardware changes (which is currently a major cost in fields such as radio astronomy, where receiver installations have relatively limited lifetimes), and they are already well-known in the field as DSPs. A complete system-on-chip (SOC) for a radar receiver, though infeasible at present, would be a very attractive option for designers in future. Such a system could perform the various tasks of DSP, data smoothing, and possible multi-track management in a highly parallel and pipelined manner, with no communication to external devices necessary.

Using FPGAs for HPC usually involves taking advantage of massive parallelism and fixed-point number formats. Although clock speeds are an order of magnitude slower than that of ASICs and conventional microprocessors, the parallelism exploited usually results in a performance gain. The true beauty of FPGAs, however, besides their potential for parallelism (where other architectures such as GPUs can outperform them) is their flexibility and high internal memory bandwidth[31]. To make the most out of this hardware, designers should move away from the idea that all calculations need to be done in standard single or double-precision floating-point format, and consider "computing-just-right". Custom precision floating-point and fixed-point formats on FPGAs can make designs more efficient and accurate than existing ones, and the topic is currently enjoying

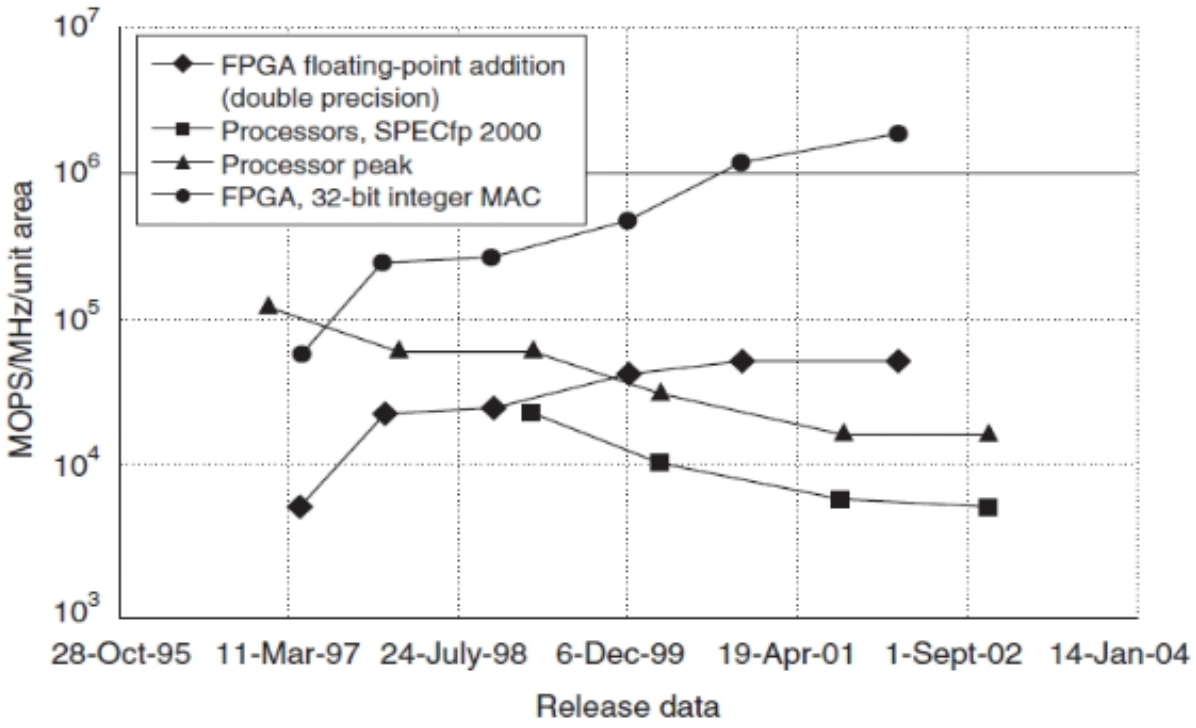


Figure 1.2: Graph comparing the performance of FPGAs and processors between 1995 and 2004[23]. While FPGA performance has been increasing, conventional processors show decreasing performance per unit area.

significant attention from researchers [31, 29, 32, 43].

Coprocessor cards and in-socket accelerators featuring the devices are commercially available (from companies such Nallatechnal [14], DRCdrc [11], and Xtremedata[8], to name a few) and facilitate the programming of the FPGA and communications between it and the conventional desktop computer.

1.2 Algorithm of Study

The algorithm investigated is computationally demanding for a number of reasons. Firstly, it is *non-recursive*, and makes use of a significant amount of data gathered over the track's history in order to produce its output. Recursive filters such as the Kalman use only the most recent measurements. Secondly, it requires very *high precision* computation, and usually makes use of floating-point numbers. Lastly (and perhaps most importantly) it is *iterative* in nature, and must cycle through the data a number of times before producing its result. This makes accelerating it by taking advantage of parallelism more difficult, and at first glance it does not appear to be ideally suited to acceleration on

FPGA architectures. However, instead of relying only on massive data- or task-based parallelism,[1] this study considers custom pipelines and reduced-precision number formats for acceleration on the hardware.

Although the original algorithm design is very general, this study focuses on a specific implementation of it: a tenth degree Doppler-bearing filter, for use in applications resembling air-traffic control. The specific implementation we investigate is made even more weighty by the fact that it takes detections from an array of eight PCL radar receivers as input, further increasing the data that must be processed. Some aspects of its design are also nonlinear, which increases its complexity.

The algorithm is discussed in more mathematical detail in section 2.2, and analysed in terms of its computational load in chapter 3.

1.3 Objectives

The main objective of this dissertation is to investigate the acceleration of a Gauss-Newton radar tracking algorithm using FPGA technology. This is divided into the following sub-goals:

- Carefully analyse the computational requirements of the algorithm, investigating the possible parallelism/efficient pipelining that can be taken advantage of for acceleration.
- Investigate the FPGA resource requirements of the various filter components, with a view to minimising this so that maximum parallelism can be exploited.
- Investigate the effect of varying the bit-accuracy used in the algorithms. What reduction from double precision can the filters tolerate, if any? Is there a bit-accuracy sweet spot, trading off accuracy and FPGA area optimally? What exactly is the nature of the precision/performance tradeoff?
- Port sections of the algorithm to a reconfigurable computing platform, with a view to accelerating it.
- Compare accelerated versions of the algorithm with a conventional one, to determine if significant speedups can be achieved and if acceptable accuracy has been maintained. Using these results, consider if any extra design effort, implementation complexity and cost is justified.

[1] Data-parallelism is made use of when a problem requires the same operation to be done to large amounts of data, task-parallelism involves performing many different, independent tasks simultaneously, usually on smaller amounts of data. An example of the former is a large matrix-multiply, and an operating system would be an example of the latter.

1.4 Scope

In this study we investigate the acceleration of an existing implementation of the Gauss-Newton tracking algorithm - making any significant changes to the underlying implementation of the theory is not considered. It may be possible to tweak the algorithm to better suit reconfigurable hardware, but this is left for future work. Similarly, the existing software implementation which was used as a speed and precision benchmark for the system was not improved upon. It is single threaded, but was compiled with a high level of optimisation and makes use of ATLAS BLAS libraries for efficient linear algebra.

All the radar data used in this project is synthetic and arguably simplistic, and is obtained from a simulator written in the IDL programming language. Testing with real-world data is left to future work.

Only a portion of the algorithm is ported to hardware. Sections are examined in a piecewise manner for a hypothetical design which will fit on to either current hardware or an FPGA of the near future. We target Xilinx Virtex-4 and Virtex-5 FPGAs with low speed grades, and although both are still widely used they are not state of the art.

An active field of study is that of custom operators for FPGAs, including coarser-grained operators than the standard arithmetic. For example. the FloPoCo project is looking at operators such as $\sqrt{x^2+y^2+z^2}$ and $\frac{x}{\sqrt{x^2+y^2+z^2}}$ [30]. Using such custom operators, or improving on existing operators for FPGAs was not considered in this study, and we restrict ourselves to the use of standard operators and libraries which are commercially available and already widely used.

Finally, a major portion of any co-processor design is the consideration of communications between host and device. In this dissertation we focus on the device-side design, that is, the section of work handled by the FPGA. Communications and the physical implementation of the complete design are not considered in detail, and the final speedup calculation is an estimate based on simulation results and existing communication options.

1.5 Related Work

1.5.1 Previous study

This study follows the work of Joseph Milburn of the University of Cape Town[52]. In his masters dissertation, the same algorithm was ported from an IDL implementation to C, profiled to determine what the most computationally intensive sections were, and then accelerated by offloading these sections from the conventional CPU to a ClearSpeed Advance X620 co-processor card. A speedup of 2.22 was achieved, although there was an order of magnitude decrease in accuracy in the accelerated implementation. The CPU implementation written in C was used as a starting point for this study,

together with the original IDL simulation on which it was based.

1.5.2 Accelerating Radar Algorithms

There has been increasing interest in applying parallel processing techniques to surveillance applications for over a decade, and much research has gone into accelerating radar algorithms. The term "radar algorithm" is a broad one, however, and can include such wide ranging examples as Synthetic-Aperture-Array (SAR) imaging (a task well suited to GPUs [33, 48]) and environment simulation [20]. The parallel tracking of multiple targets is a radar application that has attracted much attention [72, 26]. Although this intuitively seems a trivial task to parallelise, the assigning of detections to tracks proves to be quite a challenging problem.

Interactive-Multiple-Model (IMM) radar tracking algorithms are another application which lend themselves well to pluralisation. These algorithms track targets with more than one filter model simultaneously, and use a weighted sum of the different filter results as the final output. Examples of work in this area include a study making use of transputers[27], and a hardware design for implementation on an ASIC or FPGA[56].

Most research to date regarding radar tracking algorithms has considered various forms of the Kalman filer. Parallel versions of the algorithm have been shown to perform well both on FPGAs[24] and various other parallel architectures[69, 21].

Although not identical to the algorithm investigated in this study, and not applied to a radar problem, an FPGA implementation of a Gauss-Newton algorithm has been investigated by [19]. The implementation achieved significant speedup not only compared to a conventional CPU, but over a DSP as well. The hardware design was also contrasted with a Microblaze soft-core processor on the same FPGA, showing a speedup of more than 100 compared to it. This highlights the importance of making use of the flexibility of FPGAs as opposed to imitating the behaviour of processors.

Another filter known as the Gauss-Seidel Fast Affine Projection (GSFAP) algorithm has also been shown to port well to FPGA hardware[51]. Although also not radar-related, it is similar to our algorithm in that it is iterative, non-recursive, and generally requires floating-point arithmetic.

1.5.3 HPC with FPGAs

In addition to the acceleration of specific computing problems, facilitating the use of FPGAs for general HPC applications is also an active field of study. Examples of proposed FPGA-based acceleration platforms include the Nallatech in-socket FPGA front-side-bus accelerator [25], in which FPGAs are housed in processor sockets like conventional CPUs, as opposed to accelerator cards that attach to an expansion bus. This means that the devices have the same access to system memory as a CPU.

Other platforms such as the Berkeley Emulation Engine 2 (BEE2)[23], the Interconnect Break-Out Board (IBOB)[4] and the Reconfigurable Open Architecture Computing Hardware (ROACH) board[7], all developed by the Centre for Astronomy, Signal Processing and Electronics Research (CASPER)[2], are designed specifically with DSP applications in mind (in this case, radio astronomy). Although they have been used successfully in many such applications (the BEE2 being used by NASA's Deep Space Network group [3]), because their processing power is based on reconfigurable hardware they can also be used for general HPC; The BEE2 is used by Starbridge Systems[17] to accelerate Spice simulations, and fellow students at UCT are using the ROACH board to accelerate bioinformatics algorithms.

Much of the research into using FPGAs as computing platforms involves investigating means of shortening development time and abstracting away low-level hardware design details, with an aim towards eventually making them as easy to program as conventional CPUs. An Example of such work is the Berkeley Operating system for ReProgrammable Hardware (BORPH)[38], which handles hardware resources as if they were software processes on a CPU. Tools such as Nallatech's DIME-C[34], and the MyHDL project[5] allow designers to avoid explicit hardware design by converting software source code to HDL, and GUI-based design tools such as Simulink have long been used to shorten hardware development time.

For both of these reasons (the proven performance boosts, and the ever increasing ease of use), we can expect FPGAs to see more and more use as a computing platform in future.

1.5.4 Floating-Point on FPGAs

The implementation of floating-point numbers on FPGAs had been investigated even before existing hardware made it possible, and once technology caught up a host of floating-point libraries and cores quickly appeared[31]. Whereas in the past floating-point applications were avoided because of limited logic resources, many floating-point problems can now successfully be ported to FPGAs[35, 67, 46]. As floating-point numbers are so useful and ubiquitous in computing, this interest should not be surprising.

A number of options exist for designers wishing to perform floating-point arithmetic on an FPGA, for example:

- FloPoCo [29] is a floating-point core generator written in C++. The purpose of the FloPoCo project is to explore the many ways in which the flexibility of FPGAs can be exploited for arithmetic, instead of relying on operators that mimick those available in processors. Examples of custom operators available with FloPoCo are the large-accumulator for sums of products [32] and multiplication by a constant[58]. See [31] for more on the concept and philosophy behind FloPoCo.

- A set of fixed- and floating-point packages for VHDL[22] which has been in development for a number of years is now an official support library for VHDL-2008, the latest version of the language. Unfortunately, not all FPGA development environments support VHDL-2008 yet, but older versions of the packages can be used and are available from the project website[9]. The libraries aim to provide a higher level of abstraction to facilitate the use of fixed- and floating-point arithmetic, so that hardware designers will eventually be able to use floating-point numbers just as easily as they now use integers[22].
- VFloat[71] is another variable precision floating-point library for FPGAs. A notable feature of this library is that it separates the normalisation and arithmetic components of the floating-point operators, allowing for greater control during design.
- Like most FPGA manufacturers, Xilinx now includes floating-point IP cores for basic arithmetic in its COREGen IP catalogue[59].

1.6 Plan of Development

The rest of the dissertation is laid out as follows:

Chapter 2 provides background information, providing more detail on the algorithm to be accelerated, the strengths and weaknesses of FPGAs and with regards to HPC, and fixed- and floating-point number systems. We also make some comments on the concept of "computing-just-right", and describe the synthetic dataset used to characterise and test the design.

Chapter 3 discusses algorithm analysis, in which we present the results of profiling the software implementation of the algorithm, which show where the performance bottleneck is located. In this chapter we also discuss the number precision investigation that was performed, and show the reduction in precision and dynamic range that can be tolerated by the algorithm.

Chapter 4 discusses the FPGA hardware design, describing its structure and the reasons for the various design decisions made.

Chapter 5 discusses the implementation, verification, and testing of the design.

Chapter 6 presents the results of the design both in terms of its accuracy compared to the software implementation, and its FPGA resource usage efficiency. A theoretical speedup is presented based on synthesis results and current options for device-host communications.

Finally, in *Chapter 7*, conclusions are drawn from the results, and recommendations for future work are made.

Chapter 2

Background

This chapter provides necessary background for the various aspects of the study. It begins with information on the algorithm to be accelerated: data smoothing methods in general, and then more detail on the Gauss-Newton Polynomial Filter as used in the study. The specific hardware used, namely the Xilinx Virtex family of FPGAs, is then described. Background is provided on floating- and fixed-point number systems, and the concept of computing-just-right is discussed. Finally, the experimental set up of the study is described, including information on the synthetic data set used to profile and test the design.

2.1 Data Smoothing

Data smoothing is also known as "state estimation" or simply "filtering" when used in the proper context. The process of data smoothing is useful not only for remote sensing applications like radar, but in any situation where we wish to capture general trends or important patterns in data and filter out noise. Data smoothing is used in control theory, computer vision, economics, and statistical surveys, for example. It should thus be noted that accelerated smoothing algorithms would be useful for a variety of applications besides the one suggested in this study.

As the focus of this dissertation is the technical aspect of porting certain sections of the algorithm onto FPGA hardware, the mathematical principles behind all of the Gauss-Newton filter's inner workings will not be discussed in great detail. For a more in depth discussion of the theory, see [55, 54].

2.2 The Gauss-Newton Tracking Algorithm

We now describe the Gauss-Newton Polynomial Filter investigated in the study. We begin with a simple example to explain the basic underlying concepts, then contrast it with our more realistic

version in order to discuss the more complicated aspects.

2.2.1 Simple Example

The Observation Equation

We consider the problem of estimating the true state of a system based on noisy observations of it at discrete time instances. Let X_n be a vector of the states we wish to track. Let Y_n be a vector of observations made with the measuring instruments. Y_n is then related to X_n by the *observation equation*

$$Y_n = MX_n + N_n \tag{2.1}$$

where N_n represents additive noise. We include the subscript n to indicate that we are dealing with the most recent states and observations, which are part of a series in time.

As a simple example, let us define X_n to be the position and velocity of a radar target in x, y and z coordinates:

$$X_n = \begin{pmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ z \\ \dot{z} \end{pmatrix} \tag{2.2}$$

In the simplest case, we can directly observe some of the the states we wish to track, and M (which we term the *sensitivity matrix*) is then simply a matrix of the constants 1 and 0. For example, if we wish to track our states based on observations of the target's position along x, y and z axes, the observation equation is:

$$Y_n = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ z \\ \dot{z} \end{pmatrix} + \begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix} \tag{2.3}$$

The Minimum Variance Algorithm and Transition Matrix

By use of the ***Minimum Variance Algorithm*** (which we will not prove here, see [54, 55] for proofs), it is possible to obtain the statistically best possible estimate of X_n by linear transformations of Y_n based on:

- equation 2.3
- the known variances of the observations (based mainly on the specifications of the measuring instruments)
- State vectors vectors from previous timestamps in the batch (or *leg*) of data, since we wish to implement the original, non-recursive form of the algorithm.

Because of this last requirement, and because filters are also required to make predictions about future observations, we require a method by which we can project the states backward or forward in time according to some model of the target behaviour.

This is done by means of what is known as the *transition matrix*, which is based on the *filter model* (or *internal model*) of the system. The filter model is always a set of Differential Equations (DEs) which describe the change of states over time. For our simple example, the filter model might be:

$$D\begin{pmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ z \\ \dot{z} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ a & b & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & c & d & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & e & f \end{pmatrix} \begin{pmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ z \\ \dot{z} \end{pmatrix} \tag{2.4}$$

If acceleration depended on position and velocity for our system. (Note that D indicates the derivative operator).

Filter models can be either linear or nonlinear, and the complexity of the filter model determines the difficulty involved in constructing the transition matrix. For the simplest case of a linear (or polynomial) filter model as in equation 2.4, it turns out that the transition matrix consists of the coefficients of the Taylor Series expansion of the DEs, which are functions of the time interval we wish to project the state vector over.

The elements of the transition matrix are thus functions of the time interval between the state vector's current time-stamp, t_{n}, and the time we wish to project it to, $t_{\text{n-k}}$. We therefore denote the transition matrix by $\Phi(t_{n-k} - t_n)$.

By multiplying the state vector by this matrix we can obtain the polynomial model's prediction of the states at any time before or after the current time instance. Part of the Minimum Variance Algorithm

involves using it to obtain previous state estimates, and assembling these together with transition matrices and M matrices into a what is termed the *Total Observation Matrix*, denoted by T.

2.2.2 Our Case

The problem we investigate in this study is more complex than the preceding example in three ways:

1. the states of interest are the 0^{th} to 10^{th} derivatives of the radar target position in three dimensions
2. the observations are not a subset of the states themselves - they are instead Doppler and bearing observations from 8 PCL receivers
3. instead of a matrix M, we have a set of equations relating these states to the observations we obtain from radar receivers, which we denote by $G(X_n)$.

Our state vector and observation equation are thus:

$$X_n = (x, \dot{x}, \ddot{x}, \dots D^{10}x, y, \dot{y}, \ddot{y}, \dots D^{10}y, z, \dot{z}, \ddot{z}, \dots D^{10}z)^T \tag{2.5}$$

$$Y_n = \begin{pmatrix} y_1 \\ \vdots \\ y_8 \end{pmatrix} = \begin{pmatrix} g_1(x...D^{10}x, y...D^{10}y, z...D^{10}z) \\ \vdots \\ g_8(x...D^{10}x, y...D^{10}y, z...D^{10}z) \end{pmatrix} + \begin{pmatrix} v_1 \\ \vdots \\ v_8 \end{pmatrix} \tag{2.6}$$

Where the D^{n} operator indicates the nth derivative.

These equations in G describe the relationship between the observations and the states. This relationship is known as the filter's ***observation scheme***, and the complexity of tracking filters depends in large part on whether it is linear or nonlinear. In our case the equations involve square roots and trigonometry, as we are converting from Doppler and bearing readings to Cartesian coordinates; we have a *nonlinear* observation scheme.

Use of the Minimum Variance Algorithm depends on linear algebra, and we cannot use it with only this nonlinear observation scheme. We thus proceed by making use of a method of *local linearisation*.

We linearise the equations in G by effectively replacing them with their first order Taylor series expansion. We calculate the partial derivative of each equation with respect to each state, and assemble these into a new M matrix. Using this M matrix in our observation equation, and performing the Minimum Variance Algorithm as we would for a linear observation scheme, results in us making the best possible guess not of the actual state values, but of ***nominal states***, which we know are close to our actual states. We denote the nominal state vector by $\bar{X}$.

We can also obtain nominal or "synthetic" observations $\bar{Y}$ by using the nominal state vector instead of the actual state vector in equation 2.6.

If we could obtain some idea of how close this nominal state vector was to our true state vector, we could add the difference to obtain an estimate of the true states. To this end, we define the ***residuals*** to be the difference between the nominal state vector and the actual state vector, and denote them by δX:

$$\delta X = X_n - \bar{X}_n \tag{2.7}$$

We similarly define δY as the difference between the actual and synthetic observations.

We now make a claim, the proof of which will not be dealt with here (for the proof, again see [54, 55]): The Minimum Variance Algorithm works to relate the residuals δX and δY just as it does for X_n and Y_n.

The algorithm

Based on the equations and concepts discussed thus far, we can describe the Gauss-Newton algorithm as a series of simplified steps.

1. Begin with an estimate of the nominal state vector $\bar{X}$, and use it to obtain $\bar{Y}$ by equation 2.6
2. Obtain the observation residual δY by the fact that $\delta Y = Y_n - \bar{Y}$. (Y_n is given; it is the actual observations we receive from the instruments)
3. Using the Minimum Variance Algorithm, obtain an estimate of δX from the information in δY, the M matrices created from G by local linearisation, and the known variances of the observation instruments.
4. Obtain X by $X = \delta X + \bar{X}$

Because of the local linearisation used in our application of the Minimum Variance Algorithm, the final stage does not bring the nominal state to the true state, it only moves it in the direction of the true state. We can repeat the process, however, by treating X as a new $\bar{X}$ and starting again from step 1. By iterating the algorithm we approach the true minimum variance estimate.

Most of figure 2.1 below, a flow chart showing the complete algorithm under investigation for this study, should now be understandable. The highlighted blocks are the sections that make up the Gauss-Newton algorithm itself, preceded by the initialisation stage, and controlled by what is known as the Master Control Algorithm (MCA). The matrix R contains the known variances of the errors in the observation vector. The matrix S is known as the *covariance matrix*, and represents (in loose terms) our confidence in the final estimation.

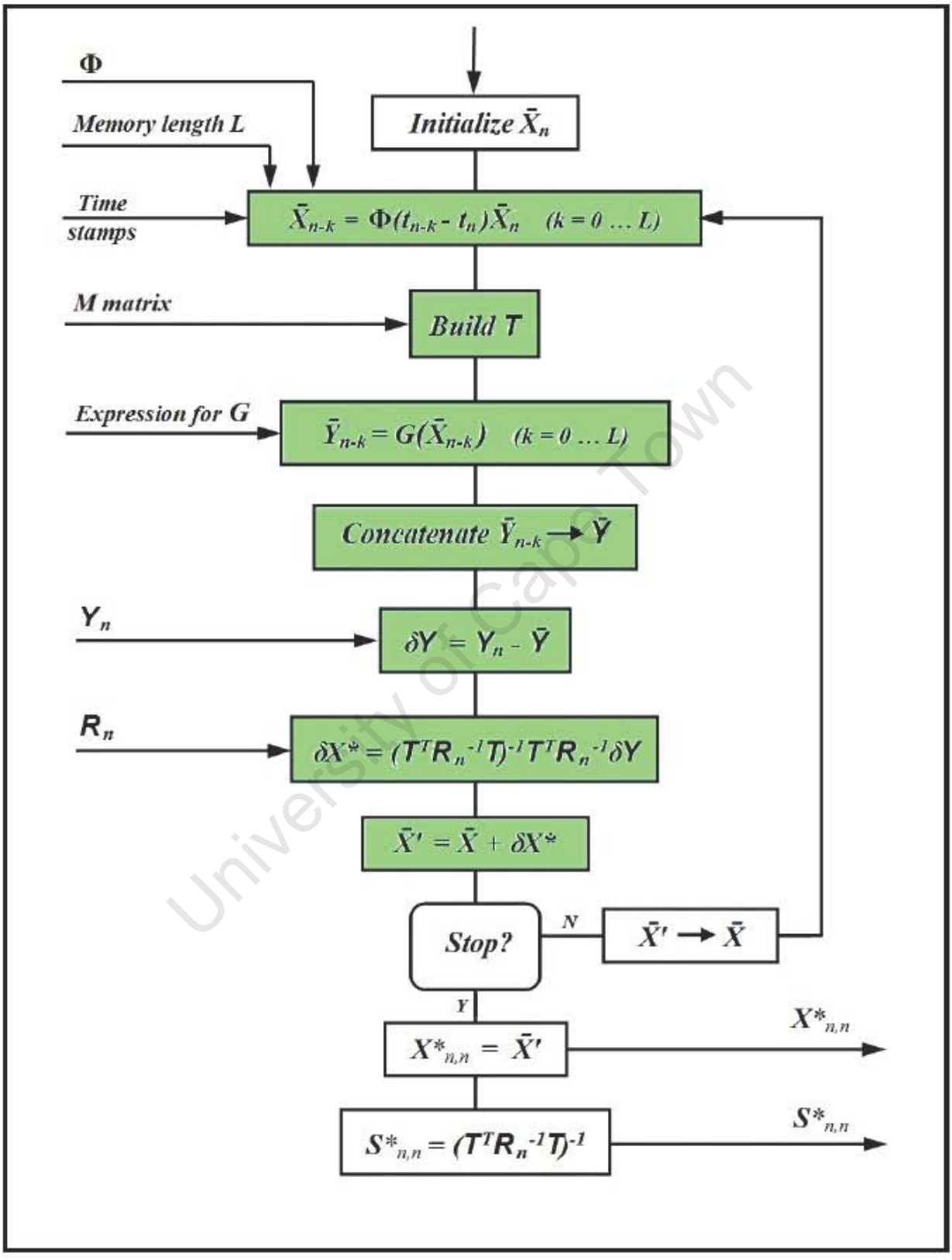


Figure 2.1: High-level view of the complete algorithm including initialisation, the Gauss-Newton algorithm, and the MCA. The construction of the T matrix is necessary for the Minimum Variance Algorithm, which is applied to obtain δX in the second last stage before the stopping rule test.

Initialisation and the stopping rule

Two details remain to be discussed: the method by which the initial nominal state vector is obtained, and the stopping rule that determines when to stop iterating the algorithm.

Initialisation consists of passing the leg's data through a much simpler prefilter. It fits a straight line to the batch of observations, and passes the starting point of this line as the initial estimate for $\bar{X}$.

The stopping condition consists of three tests applied after each iteration. If any of them pass, the state vector estimate is stored and we proceed to the next leg of data.

- **Residual limit:** the *residual vector*, δX, represents the difference between the *nominal state vector* and the actual state vector. The goal of the algorithm is to drive this as close to zero as possible so that the state vector is a good estimation of the actual state of the target. This difference is checked against a threshold value, and if it is less than the threshold this test returns true.
- **Successive estimation difference limit:** if successive estimations are very close together, it is taken as a sign that the algorithm has converged. This test returns true if the difference between two successive state vector estimations is below a constant threshold.
- **Maximum iterations limit:** If neither of the two preceding conditions are met after the algorithm has repeated a maximum number of times this, test returns true and the MCA proceeds as if it had converged. In our application this was set to 50 iterations.

The stopping rule thresholds were kept the same as the original IDL implementation. It is possible that by tweaking them (and possibly other aspects of the algorithm) better results could be obtained by the accelerated design, but investigations of this kind are left to future work.

2.3 Hardware Platform of Study

2.3.1 Xilinx Virtex Family

In this study we target FPGAs in the Xilinx Virtex-4 and Virtex-5 families. The Virtex-4 series was introduced 2004, and the Virtex-5 in 2006. The Virtex-6 range is also currently available, with the 7 series lined up for the near future.

The Virtex range is separated into 3 sub-families: the "LX" range FPGAs are designed for logic-intensive applications, the "SX" range is DSP oriented with more dedicated DSP resources, and the "FX" range includes PowerPC processor blocks for embedded processing. The "T" suffix to the

part number indicates that the FPGA includes PCI express endpoints and RocketIO transceivers for high speed serial connectivity. The Virtex-5 series also introduced the "TX" sub-family, which is optimised for very high bandwidth with more PCI and RocketIO transceivers.

Prices vary widely with volume and location, but as a general indication of relative cost table 2.1 presents some suggested prices taken from the Xilinx online store. Prices are generally much higher than that of alternative hardware: an Intel Xeon processor costs in the range of $224, and an Nvidia GeForce GTX280 about $420.

Sub-Family	LX Range		LXT Range		SX Range	
Part	LX30	LX330	LX30T	LX330T	SX35T	SX95T
Price (USD)	266.60	9313.00	356.00	14057.00	489.00	2917.00

Table 2.1: Cost estimates for Virtex-5 FPGAs, showing an entry-level and top-of-the-range example from each sub-family.

2.4 Number systems

We now provide information on number-systems used in modern processors.

2.4.1 Fixed-Point Numbers

The fixed-point number system represents real numbers in a simple, intuitive way. It consists of a series of bits representing the integer part of the number, followed by a series of bits representing the fractional part. The numbers can be signed or unsigned, with the two's-compliment system extending to include the fractional part of the number as expected.

Fixed-Point operations are similar to integer operations in hardware; the number is simply treated as an integer, and for some operations such as multiply and divide an extra shifting step rescales the output so that the radix point remains fixed.

We will use the notation $a : b$ to describe a fixed-point number format with a integer bits and b fraction bits in this study.

2.4.2 Floating-Point Numbers

floating-point numbers offer a dynamic range far wider than what is possible with fixed-point numbers of similar word lengths. Although they do this at the cost of precision[1], and make mathematical

[1] Of course, the resolution of floating-point numbers is also not uniform: when dealing with very large numbers the representable values are very far apart, and when dealing with smaller numbers the precision is unnecessarily fine. This

operations more complicated in terms of hardware, the dynamic range is necessary for general-purpose computations and a floating-point unit (FPU) has become a standard feature of modern processors. Very few programming languages lack a floating-point datatype.

Although floating-point numbers are ubiquitous in computer systems, they are poorly understood by most programmers and are mostly treated as a "black box". We now provide a basic background on the IEEE standard floating-point number format.

Figure 2.2 below shows the format of a typical floating-point number. It generally consists of:

- a bit indicating the sign
- n_e bits representing the *exponent,* which determines the range of the number. In the IEEE standard this is implicitly biased so that it represents an integer between 2^{n_e} and -(2^{n_e}- 1).
- n_m bits that make up the significant figures of the number. This part is referred to as the significand or *mantissa*[2]. In the IEEE standard, an implicit 1 is placed before a radix point at the start of the mantissa, so that it represents a fixed-point number m greater than or equal to 1 and less than 2.

Figure 2.2: Format of a typical floating-point number system with a sign bit, exponent and mantissa.

The real number r represented by the binary word is then given by

$$r = s(m \times 2^e) \tag{2.8}$$

Where s is the sign, m is the mantissa value (with the implicit 1) and e is the exponent value (with the implicit bias).

The IEEE 745 Standard for floating-point Arithmetic[40] is adhered to in almost all modern processors, and specifies a bit layout as described above, and a range of mantissa/exponent length combinations of which two have become most popular: *Single Precision* (SP), with an 8-bit exponent and a 23-bit mantissa, and *Double Precision* (DP), with an 11-bit exponent and a 52-bit mantissa.

For more information on number systems used in computing, see [36], a comprehensive tutorial on the subject.

can present problems in certain cases. For example, when accumulating many very small numbers, the running total can become so large that subsequent additions are completely ignored, as they are smaller than the precision of the total.

[2]We keep to the terms 'exponent' and 'mantissa' from this point forward.

2.4.3 Computing Just Right

Floating-point numbers offer a dynamic range far beyond what is possible with fixed-point numbers, but have three major drawbacks:

1. they come with a significant hardware cost
2. operations are much slower than integer fixed-point calculations
3. the inner workings of an FPU are complex and not entirely understood by most computer programmers

None of these are a major problem when writing software for mature, highly optimised machines with decades of development behind their FPUs - but when using FPGAs all three present a considerable hurdle to designers.

As mentioned by [31], although a specific application seldom needs the full range and precision of SP or DP floating-point numbers, with a conventional microprocessor it is easiest to simply convert all variables to a floating-point type, work with them using the highly optimised FPU (which is there anyway), and then only output the significant digits of interest[3]. When designing hardware, however, it is wasteful to use the standard floating-point formats for every operation. Custom precisions and computing-just-right can result in faster, more efficient designs with even more precision than the standard SP and DP floating-point formats. The practise of sticking to standard floating-point remains widespread however, as the alternative makes development much more complicated (even infeasible for big designs) and the skills and design effort needed are much higher.

Designs using custom exponent and mantissa widths, using the minimum number of bits for the required range and resolution but still using the tried and tested floating-point format, is a good start towards making use of the strengths of FPGA technology. It is relatively easy to implement and most existing cores and libraries are parametrisable in this way[29, 59]. Few designs make use of this, however, opting instead for the standard IEEE 745 SP or DP formats.

2.5 Experimental Setup

Dr Richard Lord of UCT has implemented both the Gauss-Newton algorithm and a simulator for data generation in the IDL programming language. This implementation was the starting point of the work by Joe Milburn[52], which in turn formed the starting point of this study.

The simulator generates synthetic raw radar observation data, based on a chosen PCL receiver configuration, target type, and flightpath (which together make up what is known as the “external model”).

[3] Scientific and engineering projects are very seldom interested in more than 5 significant digits - single precision floating-point stores 23.

The data is stored in a text file, together with the true states of the target which make up the actual flightpath.

The tracking algorithm itself is very general and can be parametrised in a number of ways, including the order of the polynomial model and other details about the filter which determine the "internal model", or the behaviour of the target as estimated by the filter. The data created by the simulator is then read by this program and an estimation of the target track is generated.

A third IDL program tests the performance of the filter by comparing the true target track with the estimations created by the filter. Figure 2.3a shows a block diagram of the system.

The more specific filter implementation was ported to C by Joe Milburn to create a faster, more efficient benchmark against which his accelerated version could be tested. It makes use of the ATLAS BLAS libraries[1] for efficient linear algebra, and represents the conventional way in which one would implement the filter on a standard CPU. It is single threaded and makes use of only one processing core - multithreading was not investigated in the study, and is not explored in this one either.

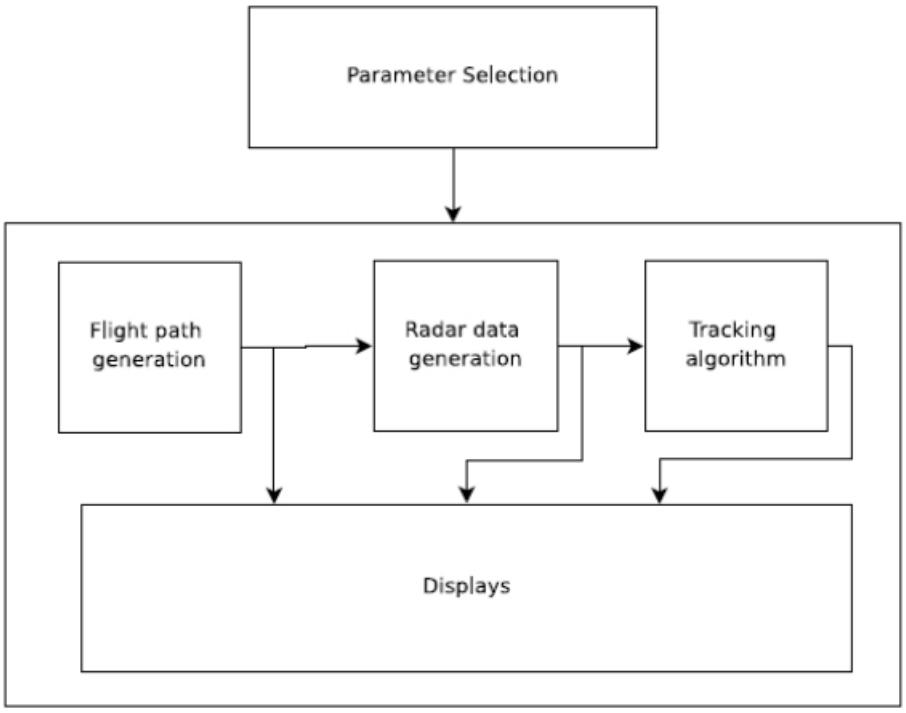
The radar data generated by the IDL code was then read by the C filter, and the results were written back to text files which were brought back to the IDL project which compares the results with the true state of the target. Figure 2.3b below shows a flow chart of the process.

It must be noted that the simulated data is generated in a fairly simplistic way, and that the system has yet to be tested on real-world or more realistic radar data.

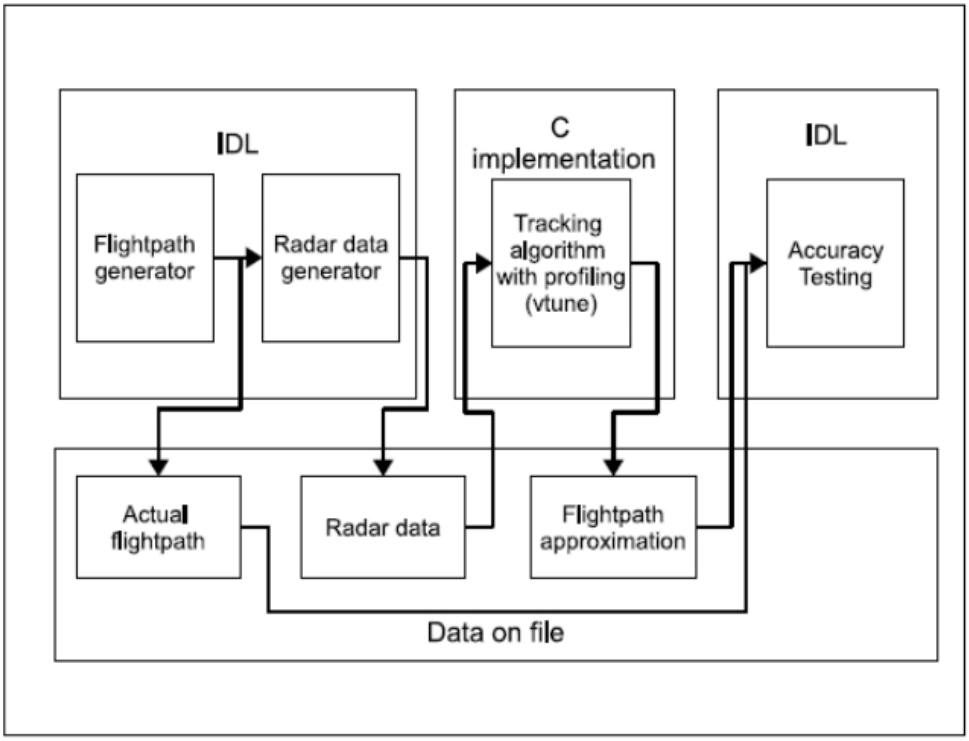
2.5.1 Dataset

A total of twenty-five synthetic data sets were created using the IDL simulator, to test the performance of the filter, and for profiling the range and precision requirements of variables. They represent Doppler and range measurements from 5 different target aircraft types each following 5 different flight paths. The aircraft differ in terms of speed, manoeuvrability, and Radar Cross Section (RCS). Plots of some of the flight paths are shown below in figure 2.4, to provide an idea of the distances and manoeuvres involved.

For all of the data used the receivers were configured in a circle with a radius of 40km as shown in the figure.



(a) Block diagram of the original IDL system[52]



(b) Block diagram of the testing arrangement used in the previous study[52]

Figure 2.3: Block diagrams showing the workings of the existing code.

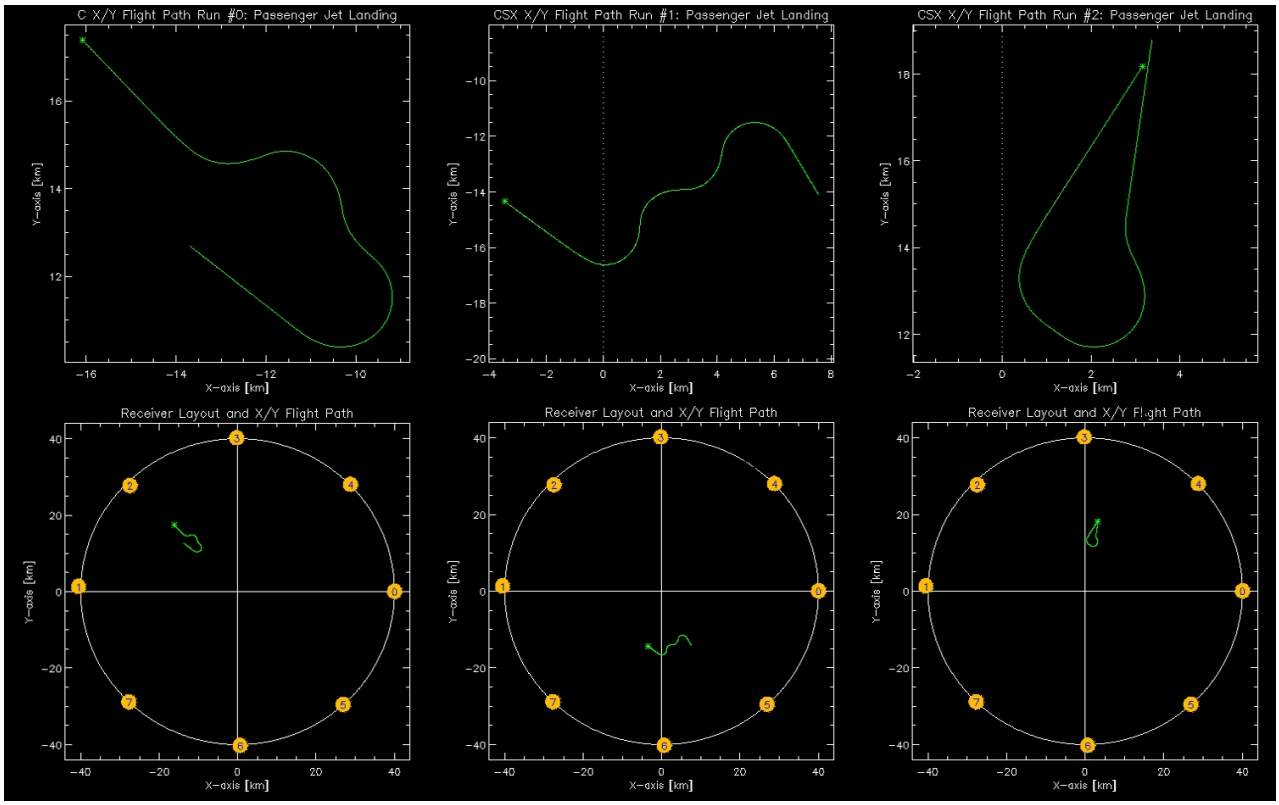


Figure 2.4: Plots of the target tracks in the X-Y plane for three simulations in the dataset. The gold circles represent the receivers.

Chapter 3

Algorithm analysis

This chapter describes the process of analysing the algorithm, which was performed in order to determine how best to attempt to accelerate it. We begin by discussing the results of profiling the C code with the Intel Vtune performance analyser tool. From these results we show the computation bottleneck to be a series of equations which calculate the partial derivatives of the radar observations with respect to the state vector. We then investigate the suitability of FPGA technology to this section of work, and proceed with the very first stage of acceleration, rearranging the equations so that they are most efficient for computation in hardware. We then discuss the results of number precision investigations, and show that the code hotspot requires significantly less range and precision than DP floating-point.

3.1 Code Profiling

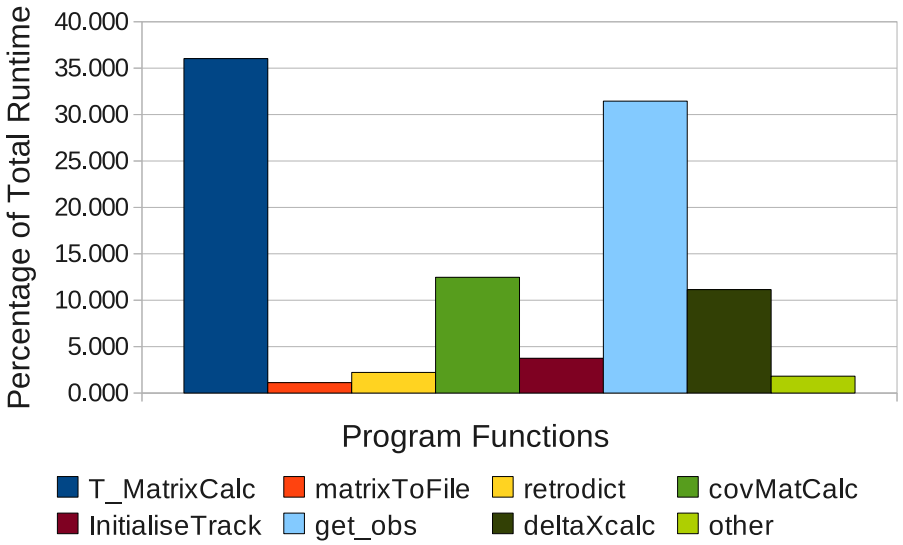
3.1.1 Finding The Bottleneck

In the work of Joe Milburn, both the IDL and the C implementations of the algorithm were profiled to determine exactly where the 'hot spot' or performance bottleneck was. Our own tests on the C code have confirmed the results obtained there. In our own tests, the code was profiled on a server with a 3GHz four core Intel Xeon processor with eight GB of RAM. The profiling tool used was Intel's Vtune Performance Analyser[57]. No changes to the source code were necessary, but the code had to be recompiled with appropriate flags linking the tool to it. The Vtune Performance Analyser offers a powerful array of features such as multiple thread profiling and the ability to create statistical call graphs, and can profile on a time-based or event-based basis. For our purposes a simple time-based profile of the program's single thread was sufficient. Another feature of the tool is the ability to combine the data from multiple runs of the program, and this was made use of to obtain the results shown. All the data sets described in section 2.5.1 were processed separately, and their profile data was then combined. In this manner a call-graph was obtained which lists the various functions of

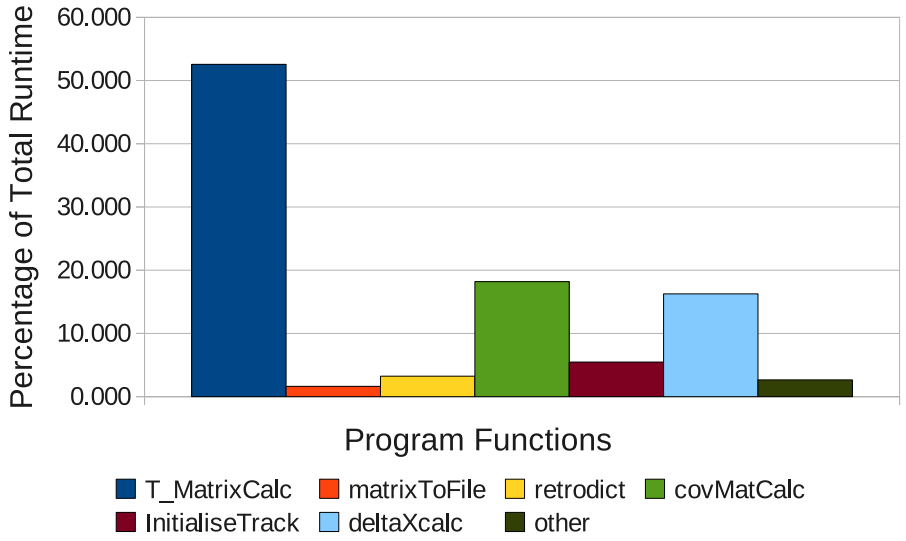
the program, their callers and callees, and a count of the number of clock cycles each function took to complete on average. The results are visualised in figure 3.1.1 below.

It was discovered that communication accounted for just over 30% with the Get_obs function taking up a significant part of the total running time. This function reads the simulated observation data for the eight radar receivers from a text file and stores it in memory. As the focus of the project was to investigate accelerating the Gauss-Newton algorithm itself, and the task of getting the data into memory would most likely be handled differently in practise (for example, DMA from an ADC), this part of the program was ignored in the profile.

Of the remaining execution time, it was not surprising to find that the Gauss-Newton differential correction (consisting of T matrix calculation, covariance matrix calculation, and the calculation of δx) accounts for approximately 90%, making track initialisation and the MCA logic negligible in comparison. Within the differential correction section, the formation of the observation matrix 'T' is most significant. This is also to be expected; it exists within two nested loops, and must be done repeatedly until the algorithm converges or the maximum limit of 50 iterations is reached, and this for each leg of the simulation. The T matrix calculation function was examined in more detail in the call graph, and it was determined that of its child functions the calculation of the M matrix was most significant within it. This function is called once for every observation within a leg (in our case, there are 80). The calculations of the partial derivatives in this function have to be done for each receiver, and account for the highest percentage of run-time within this function. Figure 3.2 presents pseudocode and a graphical display of the code sections of interest to illustrate this.



(a) Results of profiling the Gauss-Newton algorithm C implementation with the Intel Vtune Performance Analyser. Note the large percentage of runtime devoted to reading the input data from the file system (the "get_obs" function).

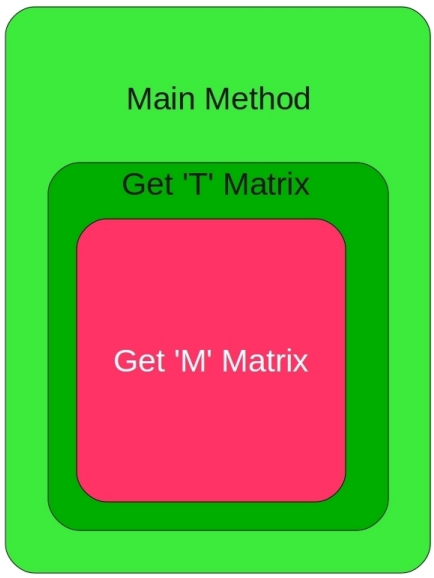


(b) Profiling results with reading and writing to the file system ignored. The three most significant functions (T_matrixCalc, covMatCalc, and deltaXcalc) are all part of the differential correction portion of the algorithm.

Figure 3.1:

```
FOR (all legs)                                              (36)
{
      while (stopping condition not met)                    (1-50)
      {
            FOR (all samples in a leg)                      (80)
            {
                  calculate results common to all receivers;
                  FOR (each receiver)                       (8)
                  {
                        calculate partials;
                  }
            }
      }
}
```

(a) Pseudo code with the computational bottleneck highlighted in red. The Gauss-Newton filter's computational intensity is made apparent by the multi-layered nested loops surrounding a significant chunk of arithmetic operations. The values in brackets in the right-hand margin represent the number of iterations for each loop.



(b) Visualisation of the percentage of total time spent in the functions of interest. The section of code to be implemented in reconfigurable hardware is highlighted in red. It represents 61% of the T calculation function, which accounts for approximately half of the total running time. The section to be offloaded to the FPGA is thus 32% of the complete tracking problem.

Figure 3.2:

It was thus determined that the place to start in accelerating the algorithm was the calculation of the

partial derivatives which make up the M matrix.

3.1.2 Initial Bottleneck Analysis

The hotspot consists of a series of equations with no branching or control logic. The equations to be accelerated are:

- Temporary variables:

$$A = (x\dot{x}) + (y\dot{y}) + (z\dot{z}) \tag{3.1}$$

$$B = \sqrt{x^2 + y^2 + z^2} \tag{3.2}$$

$$C = ((x - x_k)\,\dot{x}) + ((y - y_k)\,\dot{y}) + ((z - z_k)\,\dot{z}) \tag{3.3}$$

$$D = \sqrt{(x - x_k)^2 + (y - y_k)^2 + (z - z_k)^2} \tag{3.4}$$

$$E = \left((x - x_k)^2 + (y - y_k)^2\right)^{3/2} \tag{3.5}$$

- Calculation and accumulation of residuals:

$$\delta Y = \begin{bmatrix} f - \frac{-2\pi}{\lambda}\left(\frac{A}{B} + \frac{C}{D}\right) \\ \cos(\psi) - \frac{(x - x_k)}{D} \\ \sin(\psi) - \frac{(y - y_k)}{D} \end{bmatrix} \tag{3.6}$$

$$\delta Y_{Acc} = \delta Y_{Acc} + \delta Y \tag{3.7}$$

- The partial derivatives of Doppler with respect to the x, y, and z coordinates:

$$\frac{\delta f}{\delta x} = \frac{-2\pi}{\lambda}\left[\left(\frac{\dot{x}}{B}\right) - \left(\frac{Ax}{B^3}\right) + \left(\frac{\dot{x}}{D}\right) - \left(\frac{C\,(x - x_k)}{D^3}\right)\right] \tag{3.8}$$

$$\frac{\delta f}{\delta y} = \frac{-2\pi}{\lambda}\left[\left(\frac{\dot{y}}{B}\right) - \left(\frac{Ay}{B^3}\right) + \left(\frac{\dot{y}}{D}\right) - \left(\frac{C\,(y - y_k)}{D^3}\right)\right] \tag{3.9}$$

$$\frac{\delta f}{\delta z} = \frac{-2\pi}{\lambda}\left[\left(\frac{\dot{z}}{B}\right) - \left(\frac{Az}{B^3}\right) + \left(\frac{\dot{z}}{D}\right) - \left(\frac{C\,(z - z_k)}{D^3}\right)\right] \tag{3.10}$$

- Partial derivatives of Doppler with respect to the first derivatives of the states:

$$\frac{\delta f}{\delta \dot{x}} = \frac{-2\pi}{\lambda}\left[\left(\frac{x}{B}\right) + \left(\frac{(x-x_k)}{D}\right)\right] \tag{3.11}$$

$$\frac{\delta f}{\delta \dot{y}} = \frac{-2\pi}{\lambda}\left[\left(\frac{y}{B}\right) + \left(\frac{(y-y_k)}{D}\right)\right] \tag{3.12}$$

$$\frac{\delta f}{\delta \dot{z}} = \frac{-2\pi}{\lambda}\left[\left(\frac{z}{B}\right) + \left(\frac{(z-z_k)}{D}\right)\right] \tag{3.13}$$

- Partial derivatives of the bearing angle observations (in the form of their sine and cosine) with respect to x and y position:

$$\frac{\delta \sin(\psi)}{\delta y} = \frac{(x-x_k)^2}{E} \tag{3.14}$$

$$\frac{\delta \sin(\psi)}{\delta x} = \frac{-(x-x_k)(y-y_k)}{E} \tag{3.15}$$

$$\frac{\delta \cos(\psi)}{\delta x} = \frac{(y-y_k)^2}{E} \tag{3.16}$$

$$\frac{\delta \cos(\psi)}{\delta y} = \frac{-(x-x_k)(y-y_k)}{E} \tag{3.17}$$

where *x, y, z* are the target position coordinates and the subscript *k* is given to the positions of specific receivers. The constant λ is the transmission wavelength of the system. The partial derivatives of the bearing observations with respect to velocity, as well as the partials involving any of the higher-order derivatives, equate to zero and need not be calculated.

The required operators are thus standard arithmetic (add, subtract, multiply, divide), square root, and sine/cosine.

Initially we look for dependencies and duplication in the equations. First we note that temporary variables A and B do not depend on the target position relative to the individual receivers, and are thus common to the calculations for each receiver. We separate them from the rest of the calculations, and note that the rest (all the calculations that are receiver-specific) could be done in parallel for each receiver, and could be extended to include as many receivers as we wish.

We note that once the temporary variables have been calculated, the residual and partial equations for each receiver are also all independent, and can be performed simultaneously.

We also note that the Doppler partials (equations 3.8 to 3.13) consist of sets of identical operations performed on the three coordinates, which will simplify their implementation in hardware.

Finally we make the obvious observation that equations 3.15 and 3.17 are identical and need only be calculated once.

Note the inclusion of the residuals calculation in this function. Although these have nothing to do with the formation of the M matrix, it made sense to include them in this part of the code as they depend on the same temporary variables A to D, as well as the relative positions of each receiver, which would otherwise have to be recalculated.

We then note that it is inefficient to offload everything in the hotspot to the coprocessor - to do so means that the host lies completely idle while waiting for its results. For maximum efficiency the computational load should be as balanced as possible between host and device at all times. Unfortunately there is nothing outside of the 'getM' function that can be done in parallel on the host side, so we must choose something from within it. A detailed load balancing investigation was not carried out, but it was decided to exclude the residuals calculations from the hardware design, and treat them as part of the computation that would be handled by the host, for the following reasons:

- The residual calculations are a small percentage of the total work.[1] It was decided that excluding too little from the investigation would be safer than excluding too much, as it would be easier to give the host more work in the event of an imbalance than to design more hardware for the device. We must also consider the fact that the offloaded section will be accelerated, so offloading only 50% of the work will result in an inefficient final design.
- As mentioned, the residual calculations are independent from the partials in that they do not help to form the M matrix, and so it makes sense to exclude them rather than some of the partials, and avoid possible complications in synchronising the host and device work.
- Excluding the residuals means that we do not need to implement trigonometry on the FPGA.

Performing the residuals in parallel on the host side does however mean that there is some repetition of work - namely the calculation of the temporary variables A to D.

Once the section of code to be ported to hardware was decided upon, it had to be considered in terms of its suitability for acceleration on the chosen hardware. The following aspects were considered, each examining whether either a specific strength of FPGAs could be exploited, or a weakness avoided.

- **Can parallelism be taken advantage of?** The calculations have to be done for all observations in a leg, a number of times until the algorithm converges, for each leg. Each iteration of the algorithm depends on the results of the previous iteration, so this is unfortunately inherently sequential and cannot be done in parallel. The legs are also inherently sequential; they represent batches of data operated on in turn as they arrive from the radar system. Within each leg are eighty observations for which the calculations are independent. However, performing all the calculations in equations (3.1) to (3.17) eighty times, in floating-point, entirely in

[1] By doing a count of operators and making a rough adjustment for relative operator cost (weighting square root and division by a factor of three), the residuals account for 16% of the function.

parallel is a lot to ask of even the largest current FPGAs, so we must look for more course-grained parallelism. From equation (3.3) onwards, all of the calculations must be done for each receiver, and most of them for multiple coordinates of the Cartesian system. These are all independent tasks. Together with IPL, there is therefore much parallelism to be taken advantage of, even though the problem cannot be considered embarrassingly parallel.

- **Can it be pipelined?** This section of code can be considered a streaming or SIMD application - the exact same operations must be performed on a stream of data. There are there no branches or loops within this section of code[2], so it is ideal for pipelining. The eighty observations for each leg which we cannot parallelise will instead make up the stream that flows through the pipeline[3].

- **Is there a high computation-to-communication ratio?** The major bottleneck for most FPGA applications is external communication, and ours is no exception. Unfortunately, for our chosen section of code we observe that that both input and output bandwidth requirements scale linearly with the number of processing elements (PEs). One of the variables required for the calculations is the position of the receiver, and this will be unique for every PE. The outputs are also unique for each receiver and are not combined until much later in the algorithm. However, in this study we focus on the device-side design, and note that host-device communications as well as memory access speeds for FPGAs are an active field of research. In addition, with increasing fabric sizes we can soon expect to be able to port more work onto the device, hopefully improving the ratio of work to communication.

- **Is the problem data-intensive?** One way to answer this question is to ask: Does the problem fit in the cache of a conventional microprocessor[35]? FPGAs have a much greater internal memory bandwidth than CPUs, so to harness this strength, the problem should be too large to fit in the cache of a CPU. As a rough back-of-the-envelope calculation regarding this, we can consider that the inputs to our code section are eighty of the the zeroth and first derivatives of the three position coordinates, as well as the constant positions of each of the eight receivers. If we transfer data in SP which is four bytes wide, the inputs alone require (80*6 + 8*3) * 4 bytes for a total of 2kB. For the results of equations 3.1 to 3.17 one would need an additional 14 registers for each of the 80 inputs, resulting in an additional 4,48kB and a total of 6.48kB. The L1 cache of the processor used for benchmarking our design was 32kB, and even though we can expect many more registers to be necessary for the variables between input and output, the total will probably not exceed this. A more data-intensive problem would be better suited to FPGA acceleration.

[2]This section of code itself does exist within a set of nested loops, which is not ideal. However, now that we have chosen the section to be accelerated, we consider it in isolation from the rest of the system.

[3]In addition, as is made evident in later sections, the latency of the pipeline will also be large in proportion to the length of the data stream, so the benefits of parallelising all eighty observations are further diminished.

- **How does the data need to be represented?** Because of their flexibility FPGAs are not restricted to predefined data types and word lengths. For this reason they have been used successfully in applications where the data can be represented more efficiently than the data types of other systems allow. For example, in the bioinformatics problem of DNA similarity searching, the four different nucleotides can be represented by 2 bits, whereas on a more conventional processor the smallest data type is commonly 8 bits. In our application, where double-precision floating-point would normally be used, we investigate the use of reduced precision number formats for a more efficient design.

It should also be mentioned that although FPGAs have shown promising performance in linear-algebra applications [42, 47], conventional processors are already somewhat optimised for this by making use of vectorisation[4], and GPUs are specifically designed with linear-algebra type problems in mind. Our code hotspot is thus slightly more suited to FPGA acceleration as it is not an embarrassingly parallel linear-algebra problem.

3.2 Arithmetic Arrangement and IPL

The first step in accelerating the equations is to rearrange them so that they are calculated as efficiently as possible in hardware. This step should also be performed in software designs as significant performance improvements can be made in this way quickly and easily.

Equations 3.1 to 3.17 were rearranged so as to minimise the number of operators required, and to minimise the use of more expensive operators such as divide and square root. For single precision floating-point, divide and square root operators can take up two to ten times as much resources as multiplies[62], depending on DSP usage and latencies. As an example to demonstrate the savings achieved, equation 3.11 as initially presented to the designer is reproduced here in a simplified form:

$$a = c\left[\left(\frac{p}{B}\right) + \left(\frac{p_{rel}}{D}\right)\right] \tag{3.18}$$

Where c is the Doppler constant, p is the target position presented as an input to the component, and p_{rel} is the position relative to the receiver calculated by an earlier stage. It was rearranged and implemented in hardware as follows:

$$a = \frac{pD + Bp_{rel}}{(1/c)BD} \tag{3.19}$$

Where $1/c$ is simply another constant precalculated and hard-coded into the design [5]. This rearrangement replaces one division operator with three multiplies, at the cost of 16% longer latency in our

[4] In fact, recent studies[70] show that many quoted speedup values are exaggerated, as few researchers compare their results with code which is fully optimised on the conventional CPU.

[5] As an added bonus it so happens that $1/c$ is less than one, and thus needs no bits to represent its integer part.

final design. A similar strategy was followed throughout the design, resulting in reduced logic resource requirements and maximum potential use of dedicated DSP hardware on the FPGA (dividers generally cannot make use of DSP blocks, but multipliers can). As much ILP as possible was also taken advantage of for each component. For example, in this case the numerator and denominator are calculated simultaneously.

After making the changes we are left with the final list of operations that make up our PE and are to be implemented in hardware:

- Receive inputs:
  - Target position: x, y, z
  - Target velocity: $\dot{x}$, $\dot{y}$, $\dot{z}$
  - Position of each receiver: x_k, y_k, z_k
  - Temporary variables A and B, which are common to all PEs and will be calculated beforehand.
- Calculate temporary variables:

$$p_{rel} = p - p_k \qquad p \in \{x, y, z\} \tag{3.20}$$

$$C = \dot{x}x_{rel} + \dot{y}y_{rel} + \dot{z}z_{rel} \tag{3.21}$$

$$F = (x_{rel})^2 + (y_{rel})^2 \tag{3.22}$$

$$D = \sqrt{F + (z_{rel})^2} \tag{3.23}$$

$$E = \sqrt{F} \times F \tag{3.24}$$

- Calculate partial derivatives of Doppler with respect to position:

$$\alpha = \frac{D^3(B^2\dot{p} - A^2 p) + B^3(D^2\dot{p} - C p_{rel})}{D^3 B^3} \qquad p \in \{x, y, z\} \tag{3.25}$$

- Calculate partial derivatives of Doppler observations with respect to velocity:

$$\beta = \frac{pD + B p_{rel}}{(1/c)BD} \qquad p \in \{x, y, z\} \tag{3.26}$$

In its original form 4 integer bits would be needed. A possible avenue for further investigation is the efficiency of representations throughout the algorithm in terms of bit width.

- Calculate partial derivatives of bearing angle observations with respect to x and y position:

$$\gamma = \frac{(p_{rel})^2}{E} \qquad p \in \{x, y\} \tag{3.27}$$

$$\kappa = \frac{-(x_{rel})(y_{rel})}{E} \tag{3.28}$$

- For each receiver, return outputs:
  - Doppler-Position partials: $\alpha = \frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}, \frac{\delta f}{\delta z}$
  - Doppler-Velocity partials: $\beta = \frac{\delta f}{\delta \dot{x}}, \frac{\delta f}{\delta \dot{y}}, \frac{\delta f}{\delta \dot{z}}$
  - Bearing-Position partials[6]: $\gamma = \frac{\delta \sin(\psi)}{\delta x}, \frac{\delta \cos(\psi)}{\delta y}, \frac{\delta \sin(\psi)}{\delta y}, \frac{\delta \cos(\psi)}{\delta x}$

All changes that were made to the arrangement of equations were repeated in the software version of the algorithm, so that comparisons and speedup calculations would be fair.

3.3 Number Precision Investigation

One of the major reasons for the computational burden that radar tracking filters present is that they generally require very high precision. Common practise (as with most scientific and engineering software) is to simply implement all operations in DP floating-point format for safety. This virtually eliminates the possibility of over- or underflow, and the precision of this number system is so fine over most of its range that it is not even considered an issue by designers. The designer treats the variables as if they were real, with infinite precision, and has one less thing to concern himself with. This makes sense in a microprocessor, with a dedicated FPU that is highly optimised. In an FPGA, however, it would be a great waste of resources, and would leave the main strength of the platform (its flexibility) under utilised. In fact, implementing the design proposed in this thesis in full 64-bit floating-point would be impossible on even the most cutting-edge FPGA[7].

The section of code to be ported to the FPGA was thus analysed in detail to determine the minimum numerical precision required. Previous work by Joe Milburn [52] found that running the entire algorithm in SP was not possible. Double precision accuracy was required to ensure that the iterative correction did not diverge. However, the possibility of running only certain sections of the code at reduced precision was not investigated.

[6] recall that $\frac{\delta \sin(\psi)}{\delta x} = \frac{\delta \cos(\psi)}{\delta y}$, so there will be only three outputs representing the bearing partials.

[7] It should also be noted that the inputs to a real system would ultimately come from an ADC - and precisions of more than 16 bits for ADC outputs are rare. The entire system would thus be limited to the precision of its inputs, which would definitely be less than single precision. Even in our original IDL simulation, the data comes from text files containing numbers restricted 16 decimal digits in width.

As a start, then, all variables in the code hotspot were changed from type double (DP) to type float (SP) in the C implementation. It was found that the algorithm still converged and worked properly, with final accuracies similar to those of the all-double version.

Investigation then continued, to determine if a further reduction in accuracy was possible. Two avenues of investigation were followed: using a custom floating-point system with a reduced mantissa length, and a fixed-point number representation. Both methods were initially investigated in software simulation, and are described below respectively.

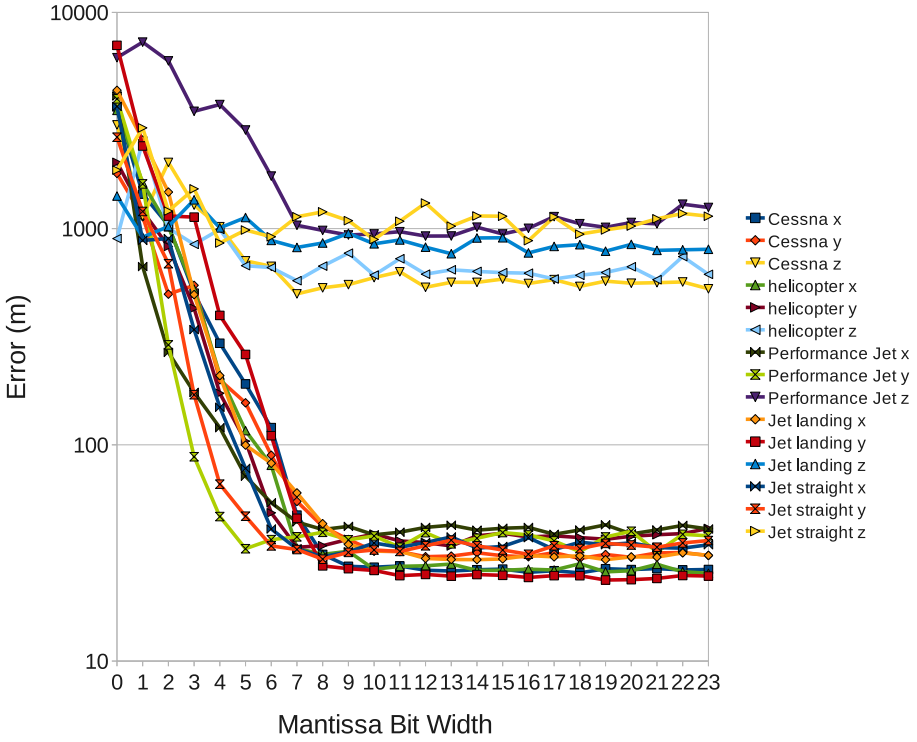
3.3.1 Floating-Point

Reduced-precision floating-point operations were simulated by making use of a simple C function, included in appendix A. It works by simply setting the last few bits of its input variable to zero, effectively "blunting" the number to a lower precision. This operation was applied to the results of all operations in the partials calculation hotspot. All tests used the standard 8-bit exponent, and reduced exponent widths were not investigated as they do not save much in terms of hardware or computation time[31].[8]. Precision and range requirements could be further examined by taking each equation or even each operator of the function in turn (using more than one word length in the design), but this was not investigated. With the help of a python script (included in appendix C), the algorithm was then run for a range of simulated mantissa lengths and the results were compared to the true state of the target to judge its performance relative to the standard single-precision version. The results are shown in figure 3.3a and 3.3b below. The poor estimation of the target altitude, z, is because of the small number of receivers in the simulation and their relatively uniform heights. The z estimations were similarly poor in the full-precision version of the algorithm. Increasing the number of receivers or arranging them with a range of different heights would have resulted in a more accurate estimation of z. Recall that a mantissa width of 23 is equivalent to SP floating-point - the rightmost result of each graph thus represents the performance of the original C code.

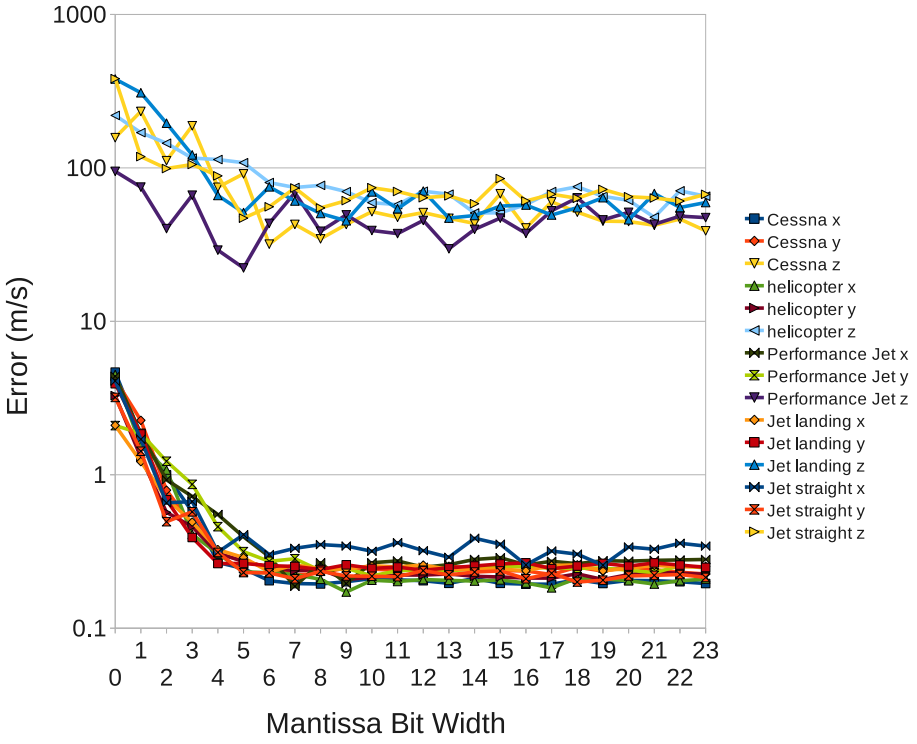
It can be seen that a clear knee exists in the graphs at a mantissa length in the region of 7 or 8 bits. Of course, when using a finite dataset to characterise the behaviour of the algorithm, there is the danger of over-fitting the design to the dataset, and for this reason all precision and range estimates had order-of-magnitude buffers added for safety (as seems to be common practise[31]). This requires 4 more bits, so for a conservative design a mantissa length of 12 bits was chosen. According to these simulation results then, 8 exponent bits and 12 mantissa bits should be sufficient for the algorithm to work exactly as it does for full single-precision all the way through[9].

[8] Later bit-true simulations of the design showed that the full 8-bit exponent was in fact necessary. Testing with a 7-bit exponent resulted in overflows, and ultimately the algorithm diverging in most cases.

[9] We can also deduce from these results that number precision was not a contributing factor towards inaccuracy in the original software implementation of the algorithm - a fact that was previously not known for certain.



(a) Average errors in position versus mantissa width. Note the knee at a mantissa length of approximately 7 bits. The estimates for z are always much worse than for x and y; this is because of the number of receivers used and the fact that they lie on the same plane on the z axis.



(b) Errors in the estimates of the first derivative. Note the similar knee at 7 bits for both x-y position and altitude.

Figure 3.3:

The number of iterations before convergence for each leg was also examined, as it was suspected that lower precision would require more iterations to arrive at the answer, potentially nullifying the final speedup of the algorithm. It can be seen from figure 3.4 that although very low precisions do indeed result in longer times to convergence, there is again a knee at a bit width of approximately 7 bits.

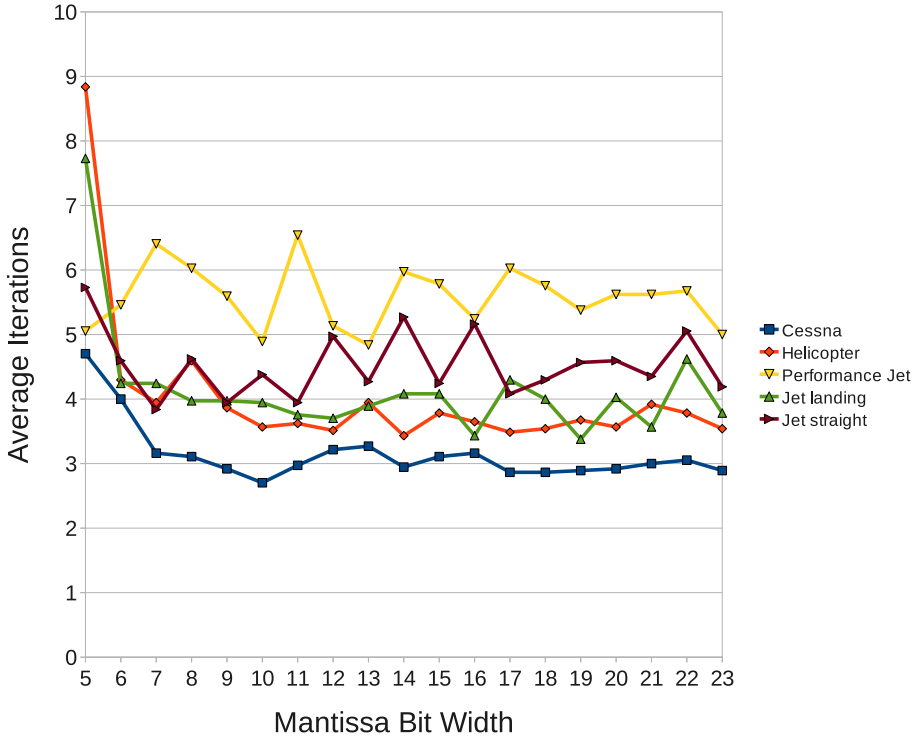


Figure 3.4: Plot of iterations to convergence for different mantissa widths. Values for less than 5 bits quickly reached 50 (the imposed maximum) and are not shown.

3.3.2 Fixed-Point

The investigations for the fixed-point word lengths required were slightly more complicated than those for floating-point. The task is not trivial; in a recent survey conducted by AccelChip, Inc. (recently acquired by Xilinx), 53% of the respondents considered converting floating-point algorithms to fixed-point to be the most challenging aspect of implementing them on an FPGA[39] (see figure 3.5).

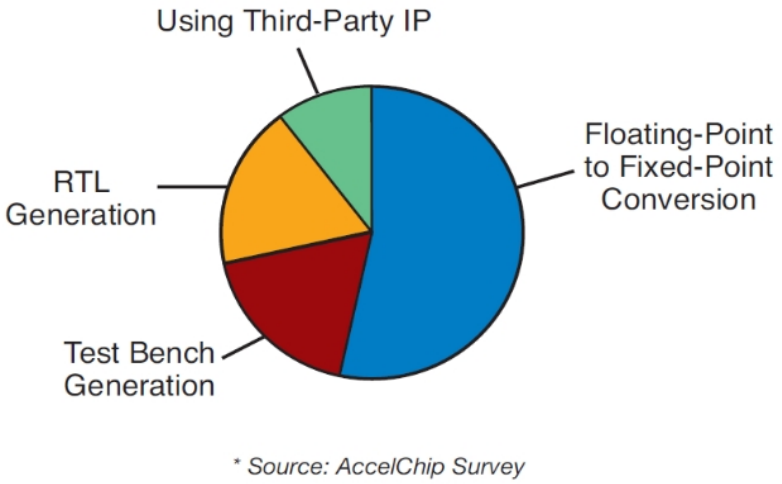


Figure 3.5: Results of a survey conducted by AccelChip, Inc. Respondents were asked to indicate which part of implementing a floating-point algorithm on an FPGA they found most challenging[39].

Proprietary tools exist for fixed-point algorithm modelling: Matlab has a fixed-point toolbox[12] and a simulink toolbox [13] which provides bit-true simulations of fixed-point arithmetic designs using words up to 128 bits wide. It also provides useful tools such as automatic overflow and saturation detection, customisable operators, and automated advisors for the conversion process.

Unfortunately these tools were not available, so alternatives were investigated. Many libraries and other resources exist for simulating fixed-point numbers of set widths[65, 68] (most are geared towards DSP programming, with 8- or 16-bit data types), but few support completely *arbitrary* word widths over the range that had to be explored. A library exists for python[16], and GNU Octave offers a fixed-point library[15], but this appears to be restricted to numbers 32 bits wide. Another option for modeling fixed-point designs is SystemC language[18]. In the end an 'in house' solution was developed, again using simple C functions, for ease of integration with the existing software written in C. It is included together with the functions for floating-point in appendix A. As the dataset used to characterise the data flow was quite large, C was also perhaps better suited to the task as it is in general much faster than Matlab or python code.

In addition to this, it was possible to characterise the dataflow through the pipeline analytically using integer arithmetic and measurements of the input ranges. The two methods and how they were used together are described below.

Software Simulation

The simulation functions convert a floating-point number to a floating-point representation of the closest fixed-point value at a given range and precision. It was assumed that any fixed-point operator implemented in hardware would similarly provide the closest possible result to the true answer. In the case of an overflow or underflow, the largest representable positive or negative value is returned

respectively.

This conversion was applied to the result of every arithmetic operation in the code hotspot, thus simulating a fixed-point system using a single word length throughout the design. One notable shortcoming of the simulation stage of the design was that only one form of rounding was investigated: round-to-zero.

A series of simulations were then run for a range of integer and fraction lengths. An exhaustive search of all integer-fraction combinations over the initial boundary estimates (0 to 40 for fraction lengths, 0 to 150 for integer lengths), would be impractical, and as it was assumed that there would be no local minima of errors, the knee of the error-precision graph was located by successively finer-grained searches.

The results showed a surprising spike just before the errors stabilise for increasing integer lengths (figure 3.3.2). After much investigation the exact reason for this remains unclear; the most likely reason is that this is a region in which only certain variables in key equations overflow, which could potentially give far worse results than if more did. For example, if just the numerator or denominator in a division overflowed, the answer could be very different from the '1' which would result if both did. This is would explain why no such spike was found in the fraction length tests, or in the floating-point investigations (overflow was not a problem with floating-point), but it is not confirmed. Discovering the reason for the spike is complicated by the fact that the final errors are the result of numerous iterations, and that for each iteration data flows not only through the reduced-precision hotspot, but through a considerable amount of conventional floating-point processing on the proposed host side as well.

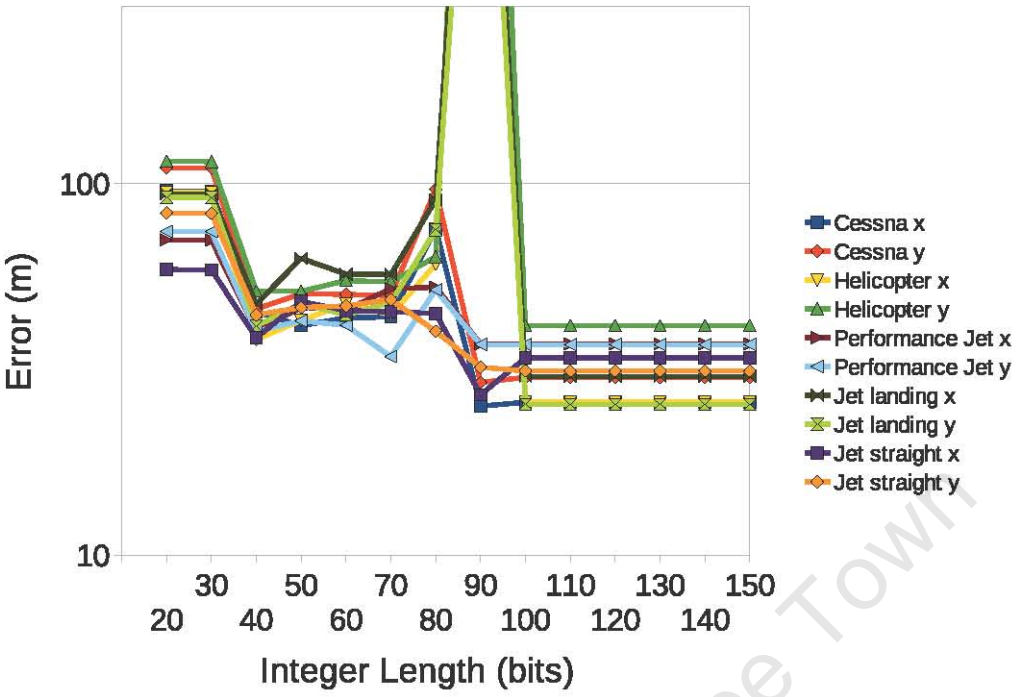
Regardless of the exact reason for the spike, it is clear that a 100 bit integer is enough for the algorithm to function properly, and that the knee of the graph sits somewhere between 90 and 100. This means that throughout our design no variable ever leaves the range $[(2^{100}\text{-}1) : (\text{-}2^{100})]$.

For the fractional part, the results showed that for lengths shorter than approximately 13 bits the algorithm could fail to converge, and the errors would sometimes diverge to infinity (or more specifically, NaNs). For lengths slightly longer than this the errors were very large (the worst of which is not shown in the graphs), but a knee was found at a length of approximately 17 bits. In cases where a few of the simulations performed worse after the knee, we considered them to be outliers and chose according to the results of the majority for the best statistical performance.

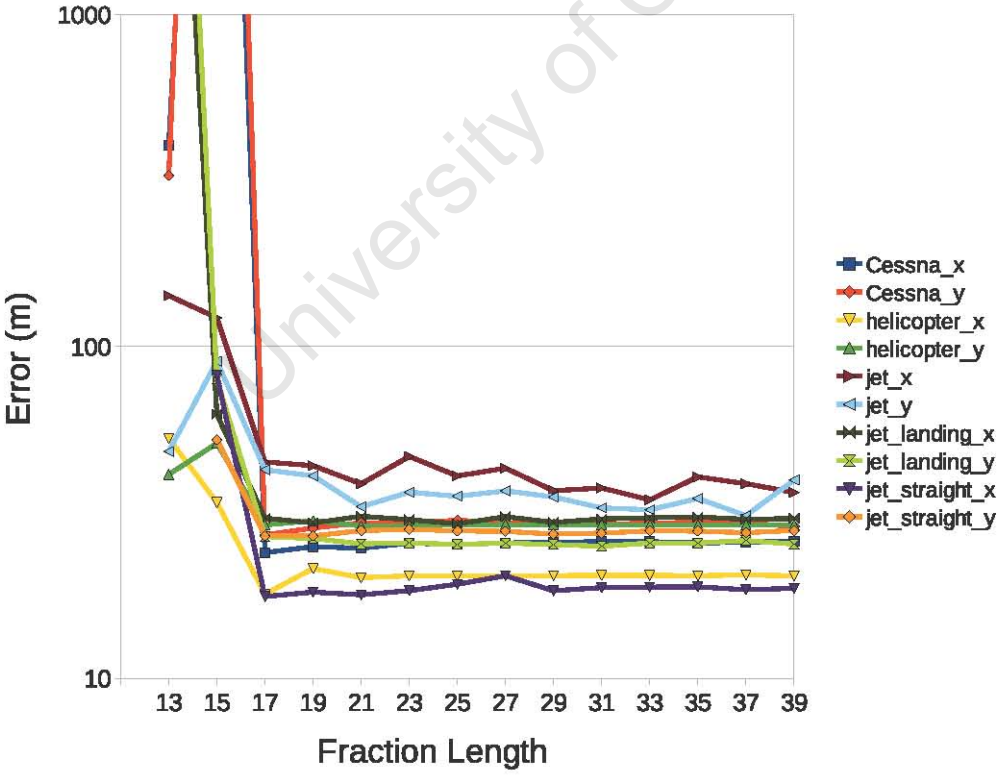
A fixed-point number system with 100 integer bits and 17 fraction bits would therefore theoretically be enough for the algorithm to perform satisfactorily.

Because these simulations used only a single number format throughout the code hotspot, we have thus far discovered only the *worst case* requirements of the pipeline in terms of range and precision. In other words, we have determined the range required by the largest variable in the equations (100 bits), and the precision required by the most sensitive (17 bits)[10]. In a fixed-hardware system where

[10]Note that the largest variable is not necessarily also the most sensitive



(a)



(b)

Figure 3.6: Accuracies of X/Y position with varying word lengths, showing the unexpected spike in the integer results for some of the simulations. At very short fraction lengths the algorithm failed to converge, and produced NaNs (not shown on graph). A subset of the results from measurements with the entire dataset is shown; for the complete results, see appendix B.

a few arithmetic units would be used and reused for all the calculations, they would have to be at least of this size, but with an FPGA implementation it is wasteful and unnecessary to use a uniform number format over an entire fixed-point design (see figure 3.8), with the only saving being development time and effort. We thus investigate further to develop a custom datapath with different word widths for the various stages in the pipeline.

Dataflow Modelling

In addition to simulation and 'blindly' profiling using the dataset, it is possible to analytically characterise the equations in terms of their range and precision requirements using integer arithmetic.

To this end, a measurement of the ranges of the inputs to the code hotspot was taken, saving their maximum and minimum values over the entire dataset to a file. The results are shown in table 3.1 below. A rough initial approximation of the required resolution was also obtained by storing the smallest value greater than zero.

	x	y	z	$\dot{x}$	$\dot{y}$	$\dot{z}$
Minimum	-2.28e4	-2.35e4	-1.43e4	-97	-93.6	3.31e3
Maximum	1.92e4	1.89e4	2.56e4	95.9	104	2.56e3
Smallest Positive	0.131	1.79e-3	3.29e-3	2.69e-4	7.01e-4	3.81e-4

Table 3.1: Maximum and minimum values of input variables to the code hotspot using the profiling dataset, along with the smallest positive value as a rough estimation of required precision.

From these results we can determine that we will need at least a format of 16:10 for the position inputs[11], and 13:13 for the derivative inputs. These values were verified by applying the conversion functions used in the earlier simulations to only the inputs, and testing the code on the dataset once again. Knowing the required input word sizes then served as a starting point for modelling the rest of the pipeline.

It is a well known fact that to prevent overflow and retain precision with integer or fixed-point numbers, *the result of a hardware multiply is twice as wide as its inputs*. This rule of thumb is a subset of a more general sizing rule for fixed-point multiplication, and similar rules exist for all operators. They are listed in table 3.2 below. By following these sizing rules throughout the design, one can be confident that overflows or underflows will never occur, and that the input precision will be maintained.

These rules were programmed into a spreadsheet to model the dataflow. In this way various input word lengths could easily be experimented with and their effect on later stages could be examined conveniently.

Once the datapath was characterised in this way, the input widths obtained from table 3.1 were entered, and the spreadsheet provided the necessary word widths for every variable in the pipeline.

[11] In twos compliment, 16 integer bits cover the range [-32768,32768], and 10 fraction bits have a resolution of $9.77e^{-4}$

Operation	Output Integer Width	Output Fraction Width
Add/Subtract	$\max(A_{int}, B_{int}) + 1$	$\max(A_{frac}, B_{frac})$
Multiply	$A_{int} + B_{int}$	$A_{frac} + B_{frac}$
Divide	$A_{int}+B_{frac}+1$	$A_{frac}+B_{int}$
Square Root	$A_{int}/2$	A_{frac}

Table 3.2: Table of number sizing rules, showing the output fraction and integer widths of operators A and B with integer lengths A_{int}, B_{int} and fraction lengths A_{frac}, B_{frac}. [9]

All operators besides square root have results wider than either of their inputs, resulting in rapid bit-growth along the pipeline and the largest words right at the end of the equations. These unfortunately are the inputs to either division or square root cores, the most resource hungry elements of the design[12]. Figure 3.7 demonstrates this by showing the word sizes at each stage in one component in the design.

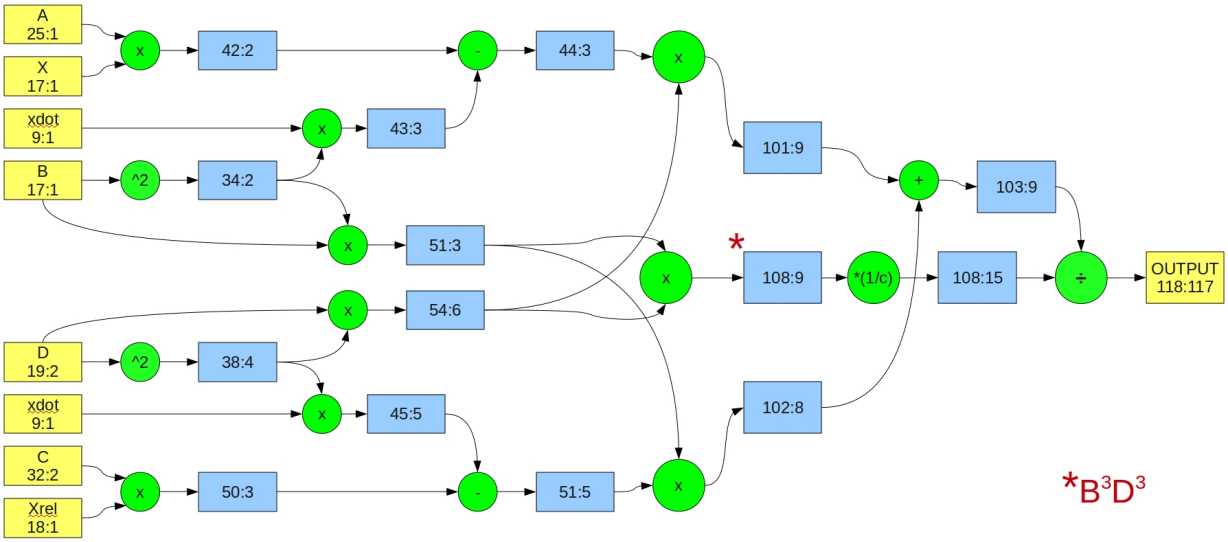


Figure 3.7: Bit growth in the calculation of the Doppler-position partials. The inputs to the component are highlighted in yellow on the left, and their respective sizes are based on preceding stages in the design. Note the unfortunate result that the most expensive operator (divide, on the extreme right) deals with the largest word sizes in the component.

As an additional test of the model, it was determined from the spreadsheet that one of the largest variables in the pipeline is the B^3D^3 term marked in figure 3.7, with an integer length of 108 bits. This value makes sense if we recall the result for maximum range obtained by software simulation, and compare it with a likely value for B^3D^3: if our position inputs can be of the order of $3e^4$ metres, both B and D will be of order $5e^4$, and B^3D^3 will be of order $1.5e^{28}$ (See equations 3.2 and 3.4). this requires 94 bits to represent, which sits in the knee of the graph in figure 3.6a.

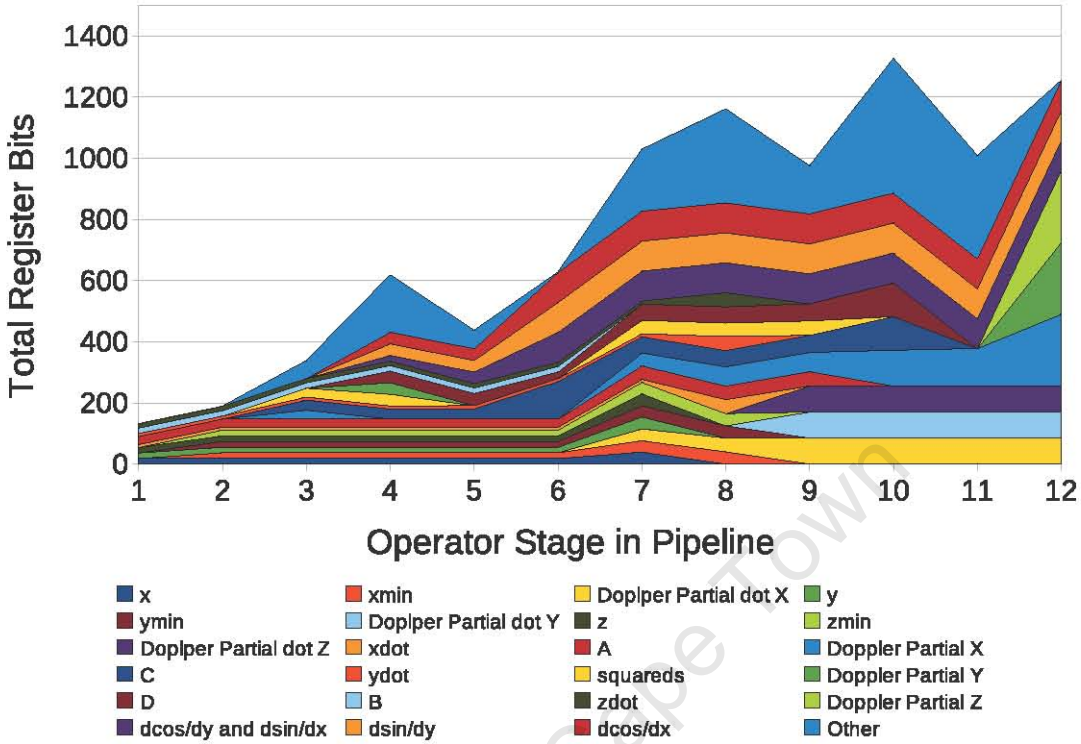
[12] See section 4.3 for more on this.

Note that strictly following sizing rules is a safe, worst-case design. There may be correlations between variables which mean that certain ranges are in fact never reached, and these correlations are only apparent when considering the equations as a whole. We also know from the simulation results that our maximum integer and fraction lengths are smaller than the maximums reached in the dataflow model, so when the word widths reach this length we can safely shorten them to the lengths determined by software profiling.

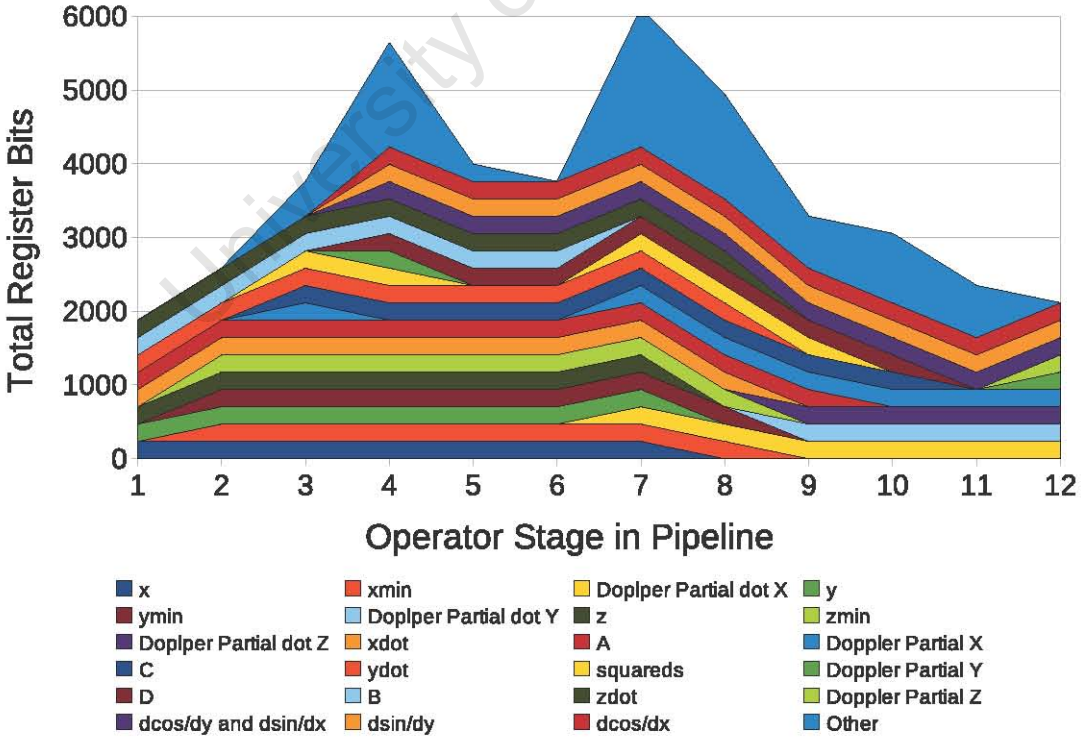
To show the savings made by using multiple word lengths in the design, Figure 3.8 contrasts the resource requirements of a flexible number format system with that of a design using the maximum word size throughout. The total number of register bits required to store all the variables at each pipeline stage are shown. The resource usage of the hardware implementing the operations themselves will also rise with rising input word size, making a flexible word size even more beneficial.

3.4 Conclusions

The code profiling confirms the conclusions of previous work, namely that the performance bottleneck of the algorithm is the calculation of the partial derivatives that make up the sensitivity matrix M. Although this bottleneck section of code is perhaps not perfectly ideal for acceleration using reconfigurable hardware because of its low communication to data ratio and the fact that it sits within an unrollable loop, there is significant parallelism that can be taken advantage of, and by considering it a streaming application we can create a deep custom pipeline for high throughput. In addition, software simulation revealed that the algorithm can tolerate a surprising reduction in dynamic range and precision over this bottleneck portion of the code. In simulation at least, a floating-point number system with an 8-bit exponent and a 12-bit mantissa is sufficient. For fixed-point numbers, simulation showed that a uniform number system with 100 integer and 20 fraction bits performed satisfactorily, and by making use of interval arithmetic a pipeline with multiple word sizes is proposed in order to further reduce hardware requirements.



(a)



(b)

Figure 3.8: Comparison of the total register bits needed to store the variables at each stage of the pipeline. Figure A shows the final design with flexible word widths, peaking at under 1500 bits, figure B shows a design using the maximum width throughout, peaking at over 6000.

Chapter 4

System Design

This chapter describes the proposed hardware designs, and the process by which they were created. It details the various design choices made and the methods used to assist in making them. It begins by discussing of the underlying concepts common to both designs, such as parallelism, pipelining, and efficient use of the dedicated FPGA hardware resources. It then proceeds to discuss the aspects unique to each design, dealing with the floating-point and fixed-point designs in turn. Finally, it presents the results of investigations into an alternative method of implementing the design using a VHDL fixed-point library.

4.1 Introduction

The equations were ported to hardware using arithmetic operator cores created with Xilinx CORE Generator[59], which is part of the ISE software toolkit used for programming Xilinx FPGAs. The cores are all fully pipelined, with parameterisable latency. The designs are both pure hardware architectures without the support of softcore microprocessors, and only a single clock domain was necessary.

Although the number precisions determining word lengths were decided upon early in the design process, both the fixed-point and the floating-point designs were implemented with parameterisable word lengths, both for safety in the event that the precision simulations proved inaccurate, and for ease of later investigation into the effect of word length on the final design size. This parametrisation was achieved for floating-point by simply storing the exponent and mantissa sizes in a VHDL package included in all the source files, and defining all relevant signal and port widths in terms of these. For the fixed-point design, the signals and port widths for every stage are calculated in terms of the widths of the preceding stage, according to the sizing rules given in table 3.2. Functions implementing the sizing rules were placed in a VHDL package, and the signal definitions which made use of them were also grouped together in this same package, in order to isolate the task of conforming to the sizing rules as much as possible from the structural definition of the hardware. Constants such as

the Doppler constant used in calculating the Doppler partials, and the smallest representable value of the number system (to be used as a substitute for zero in division denominators) are also dependent on the word lengths and are calculated and stored in this package.

Although this successfully parametrised the HDL source code, for every change in word size the IP cores had to be regenerated using the Xilinx COREGen software. This would be a tedious and time consuming process if done manually (especially for the fixed-point design, where most of the operators required a unique core definition due to the changing word widths), but it was easy to automate the task using python scripts (see appendix C), as the core parameters are stored in text files.

Figure 4.1 below displays a top-level view of both designs. Inputs are received in SP floating-point from the host, converted to either reduced-precision floating-point or the initial fixed-point formats, passed through the pipeline, and finally the outputs are converted back to SP and passed to the host.

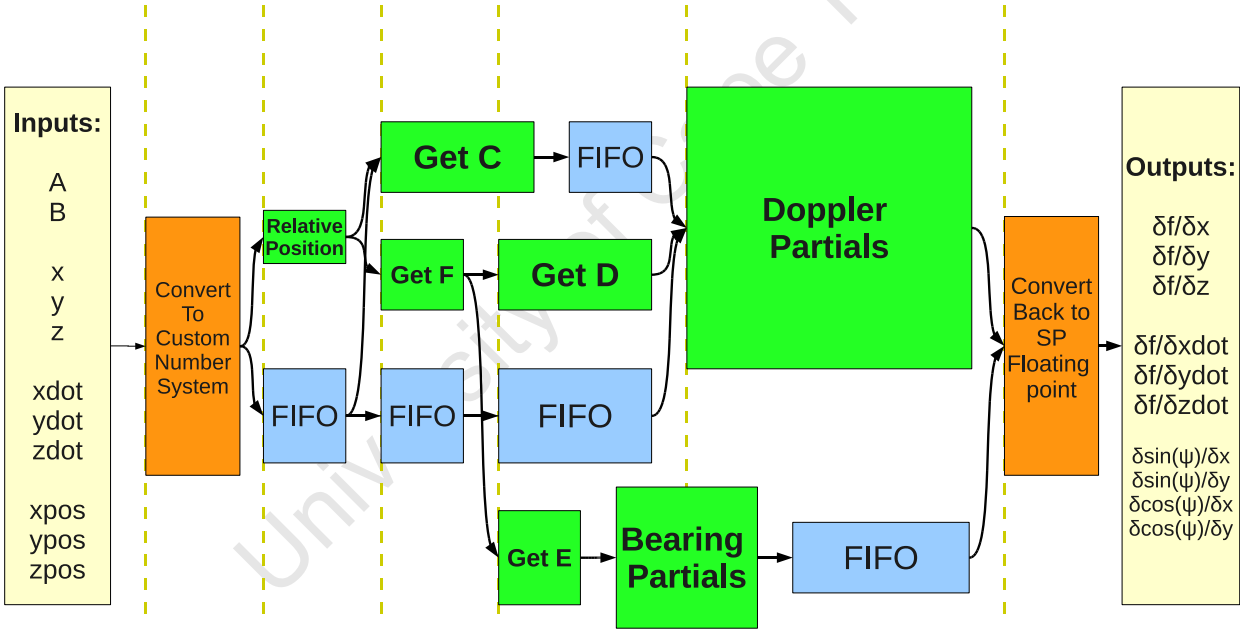


Figure 4.1: Diagram of the components making up the hardware design, showing their positions in the pipeline and their relative size based on number of operators.

4.1.1 Parallelism

The design makes use of task-based parallelism which can be visualised in terms of three levels; first, the state vector must be processed independently for each receiver, and this represents our basic PE which can be duplicated for each receiver in a given PCL system. For each receiver the same operations must be performed, the only difference being the constant values for the specific receiver's

position in x, y, and z coordinates for the calculations[1]. The second level of parallelism exists within each receiver's hardware, where much of the work to be done consists of almost identical calculations to be carried out for each of the x, y, and z components of the target's position. Below this, as much ILP as possible is implemented for the operators in each of the equations. A simplified view of the component hierarchy is shown in figure 4.2, below to illustrate this. In addition note that the hardware making up the arithmetic cores themselves is as parallel as possible, although this is the case with conventional hardware such as CPUs as well. In cases where this parallelism could be controlled in our design (see section 4.2.1 below), it was always set to its maximum.

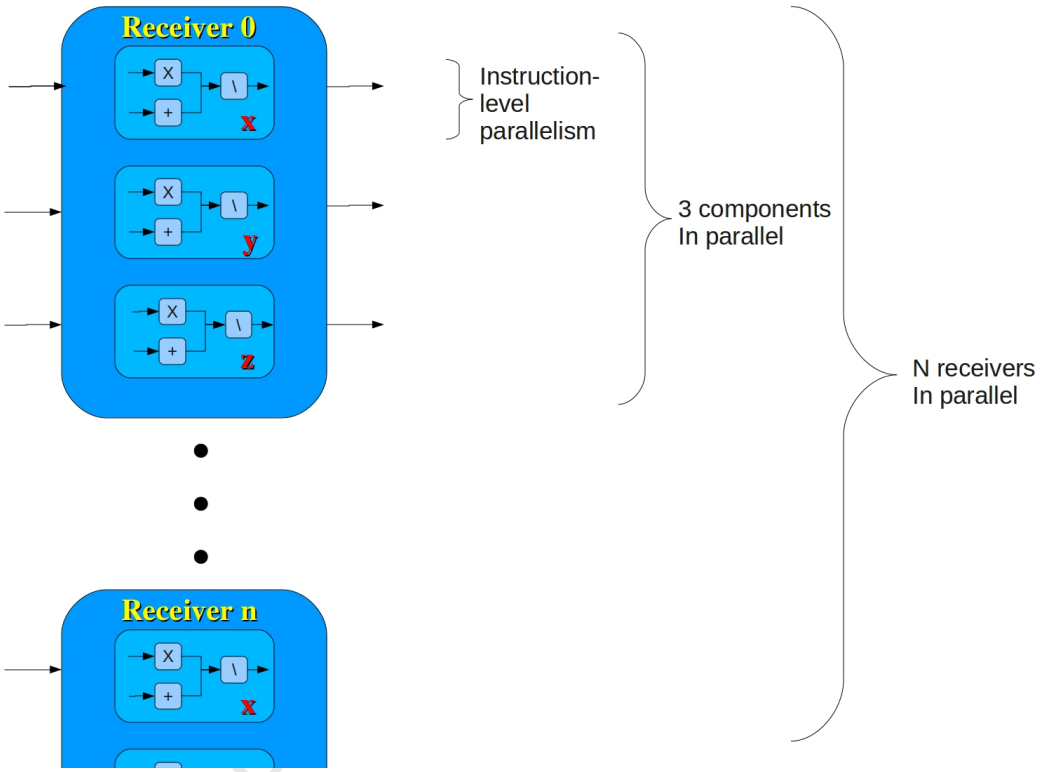


Figure 4.2: Three levels of task-based parallelism in the design: each instruction, within each position coordinate, within each receiver is parallelised.

4.1.2 Pipelining

All of the arithmetic cores used in the designs are fully pipelined. As the critical path of the design consists of these arithmetic cores strung together in series, the entire design can be considered one long pipelined from start to finish - a very deep pipeline with over 100 stages. Conventional CPUs would become very inefficient with pipelines this long, as control logic decisions such as branching

[1]For ease of testing and demonstration, these receiver position values are taken in together with the other inputs for each sample in our design. In a practical system they could be hard-coded as they would remain constant, or be taken in through the input ports at start-up and saved at the cost of eight word-sized registers and a few extra clock cycles before the algorithm commenced.

disrupt them and require the whole pipeline to be flushed before the program can continue[2]. However, in our case we are free to make the pipeline as deep as we wish (within reason); we know the pipeline in our design will never be stalled in this way, and it was found that the pipeline depth has negligible affect on achieveable clock speed.

Since our design is to be fully pipelined it is important to determine the critical path of the work to be done, so that this can be optimised for minimum overall latency, and so that areas not on the critical path can possibly be optimised in other ways such as hardware resource usage, at the 'free' expense of latency. The Gantt chart in figure 4.3 below shows the critical path of the design. It is based on latencies for SP floating-point cores, and it was assumed that the latency ratios between cores at lower precision and between fixed-point cores would be similar.

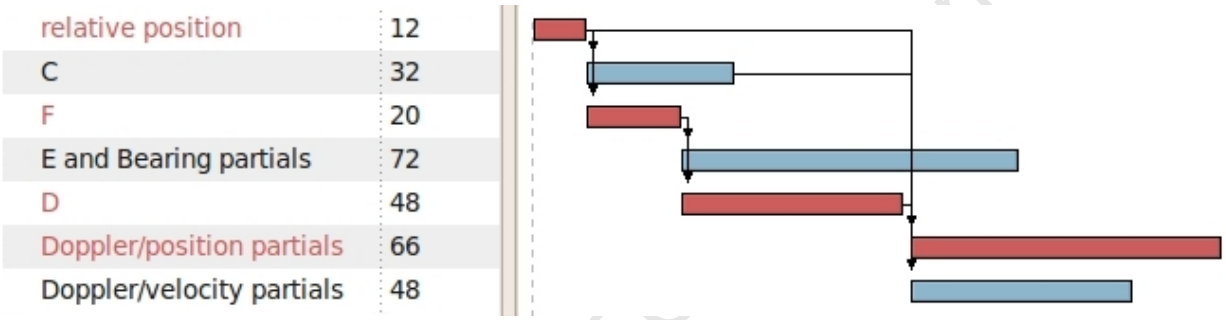


Figure 4.3: Gantt chart of the various components of the design. The critical path is in red, and the given values are latencies when implemented using Xilinx floating-point cores at SP. The calculation of A and B is not shown, as these are common to all receivers and will be calculated outside of the parallel PEs.

4.1.3 Shifting

A fully pipelined design means that a significant proportion of the hardware will have to be dedicated to shifters, so that the various parallel branches of the data stream are kept synchronised. Two options were investigated for this purpose: a FIFO generator core, and a RAM based shift register, both available in the Xilinx ISE IP core catalogue.

To compare the two, the two floating-point designs were implemented and their synthesis results compared. One design used only the FIFO core, and one only RAM shifters. The results are shown in table 4.1 below, showing that the FIFO design was more efficient. It was thus the core used.

Another advantage of the FIFO core is that it allows for simpler implementation of a parameterisable design. As the latencies of the arithmetic cores depend on their input widths, experimenting with

[2]In [37], a detailed variable depth pipeline simulation tested 35 different workloads to find their optimum pipeline depth. Most had a result between 21 and 24 stages. The results also indicated that the more modern workloads had deeper optimum pipelines.

	RAM shifters		FIFO shifters		Total
	Used	Utilisation	Used	Utilisation	Available
Slice Registers	20635	29%	17797	25%	69120
Slice LUTs	14589	21%	13107	18%	69120
Fully used LUT-FF pairs	12039	51%	10292	49%	23185

Table 4.1: Synthesis results comparing designs using FIFO and RAM-based shifter cores. The target device was a Xilinx Virtex-5 XC5VLX110T and all synthesis options were set to default. Note these results are only for a single processing core, of which there would be one for each receiver.

different sized words at various points in the dataflow can become tedious if RAM shifters are used, as the latency of the core is fixed and has to be changed manually for each design. With the FIFO, read and write ports are simply connected to the control signals of preceding and following stages, and the design works for any combination of latencies in the arithmetic cores. The minimum size of a FIFO in depth is 512 bits, and any stage of the pipeline will have a latency well below this (in fact, the maximum latency of the entire design is less than 512 clock cycles) so there is no danger of losing synchronisation due to FIFO latency limits. There are however points in the pipeline where it was necessary to specify shift latencies explicitly, as arithmetic control signals did not exist on both sides of the shift. For this purpose the RAM shifter was used to shift a single control bit, and this was then used to control FIFO shifters.

Registers to be shifted were grouped together as much as possible to facilitate sharing of the FIFO primitives. For example, in the top level view of the design (figure 4.1), the FIFO blocks shown represent the shifting of not one variable, but groups of as much as 11. Instead of instantiating a FIFO core for each variable, signals were concatenated and passed to one wide FIFO. A comparison of floating-point designs combining signals in this way with ones instantiating separate FIFOs for each variable showed that FIFO usage decreased from 71 to only 9 being used (an 87% saving), with an additional small saving in logic.

The FIFO core offers a feature known as first-word-fall-through. With this enabled, there is no latency in a read operation and the control of the pipeline is simplified. First-word-fall-through was enabled for all FIFO shifters in order to reduce latency and simplify control logic.

For the fixed-point design, the arithmetic cores could be set to have their latencies longer than the optimal recommended latency. it was assumed that this simply added registers to the output of the optimal design, and so all fixed-point cores on the same stage of the pipeline simply had their latencies set equal to the core with the longest optimum latency. In this way the design was simplified and shifting resources were saved.

4.1.4 Number Format Conversion

The Xilinx Floating-Point Operator v5.0 core[62] features an operator for conversion to and from various floating- and fixed-point number formats. This core was used in both the floating-point and fixed-point designs. Alternatively, the conversion could be done in software on the host side, but as the conversion cores are relatively small and have short latencies it was decided that it could be implemented in hardware to further reduce work on the processor side. Once the custom data path design was completed, a wrapper component with converters for all the inputs and outputs of the system was simply added and made the new top level component of the design.

4.1.5 Use of DSP Blocks

For an efficient FPGA design it is important that as much of the dedicated hardware resources on the device as possible are used. The major dedicated resources available on the Virtex FPGAs used in this study are the FIFOs and the DSP multiply/accumulate blocks.

As the available logic and DSP resources can vary widely between FPGAs (even those of the same family), the design was made as parameterisable in its logic/DSP ratio as possible, opting for IP cores which allowed for control of their DSP usage.

In general implementations of the proposed designs there will be more operators than available DSP blocks, and most of the arithmetic will have to be performed using the reconfigurable logic of the FPGA. A decision then has to be made as to how the available DSP blocks are to be distributed. In distributing DSP blocks, they should be placed on the critical path, so that the latency reductions benefit the design as a whole. Assigning them to operators with wider operands can also improve the achievable clock speed.

The main benefit of making use of the DSP blocks in our designs is that they free up logic which would otherwise have to implement the operations. Their other benefits, such as latency reduction and faster clock speeds, are made less significant when the majority of the operations are implemented with FPGA fabric.

In Virtex 5 devices the dedicated DSP resources have a fairly course granularity: they are 25x18-bit multiply-accumulates. Although many multipliers are easily cascaded to form bigger ones, one multiplier cannot perform two different multiplies at once, and thus any multiplication where the inputs are not multiples of 25 or 18 bits wide results in inefficiency. It is noted in[31] that even a Pentium processor has finer multiplication granularity than this, and since the main strength of reconfigurable hardware is supposed to be flexibility, finer-grained DSP hardware should be investigated for future FPGAs.

4.2 Floating-Point design

4.2.1 Xilinx Floating-Point Cores

The Xilinx Floating-Point Operator Core v5.0[62] was used to implement the arithmetic for the floating-point design.

Another option considered was the floating-point VHDL library by David Bishop[9], but it was found that while using VHDL packages which allow use of 'float' type variables greatly simplified design, the resource usage was too high - orders of magnitude more than when using cores. The designer also has less control over aspects such as pipelining and DSP usage.

The floating-point core supports the following operators:

- standard arithmetic (multiply, divide, add/subtract)
- square root
- comparison
- conversion to and from fixed-point
- conversion between floating-point types

Which encompasses everything needed to perform the partial derivative calculations in section 3.1. All the operators are fully synchronous with only one clock, and fully pipelined. They can be customised in terms of word length, latency, and interface. The multiply and add/subtract operators can also be customised for usage of the FPGA's dedicated DSP hardware, though only if standard SP or DP is used for the case of add/subtract.

The core is compliant with the IEEE-745 floating-point standard, with only a few minor deviations[3]:

- **Non-Standard Word lengths:** The core supports number formats with exponent and mantissa sizes outside of those specified in the standard.
- **Denormalized Numbers:** The IEEE standard extends the range of small numbers that can be represented by allowing for *denormalized numbers.* When the exponent is zero, the implicit 1 before the mantissa becomes zero, and the mantissa can represent numbers smaller than one. Such numbers are extremely small (in SP they are less than 2^{-126}) and very rarely needed. The core treats numbers that would normally be denormalized as zero, and the datasheet mentions that an increase in dynamic range can be achieved on an FPGA using fewer resources by simply increasing the width of the exponent[62].

[3] Note that similar simplifications are made in GPUs [10].

- **Rounding Modes:** Only the default rounding mode ('Round to Nearest') specified in the standard is supported. Four additional modes are included in the standard. Although it is the most accurate, this rounding mode needs an extra adder in the critical path.
- **Quiet NaNs:** If a NaN (Not a Number) is given as one of the inputs to the core, the output will be a NaN, and no exception will be raised. The standard requires both 'Signaling NaNs' (where invalid operation exceptions are raised) and quiet NaNs. There are a total of five different exception types in the standard: invalid operation, division by zero, overflow, underflow, and inexact. The cores do not handle these differently, and simply produce NaNs at their output for any exception case.

All cores can be parametrised in terms of latency, allowing designers to explore a tradeoff between resource usage, clock speed, and latency. For our designs, all were set to maximum latency in order to maximise the potential clock speed and to reduce the hardware resources needed, as the ability to fit more operators on the FPGA was considered worth the cost of extra latency cycles in a fully pipelined design. The deep pipeline also makes performance very sensitive to clock speed.

While most of the operators are able to produce an output at every clock cycle, the divides and square roots have a "cycles per operation" parameter that can be set in order to reuse hardware and save resources. In all instances, both were set to produce an output at each cycle - this resulted in a fully parallel component that consequently required more resources, but as one of the goals of the design was to exploit as much parallelism as possible this was seen as an acceptable tradeoff. According to the core datasheet, increasing this to two operations per cycle would reduce the resource usage of the cores by approximately half[62], but in our design would shave less than an estimated 5% off of the total resource usage, and because the pipeline needs to remain synchronised this would also slow down the entire design by a factor of two. An experimental design with two clock domains was investigated, where the divide and square root components were clocked at twice the frequency of the rest of the design. As expected, this simply halved the speed at which the design could be clocked.

4.3 Fixed-Point Design

Although not part of a combined package as the floating-point cores are, Xilinx offers an array of cores useful for fixed-point design including:

- add/subtract
- multiply
- multiply-accumulate

- multiply-add
- divide

All the fixed-point cores used in the design actually perform integer operations, and the hardware does not explicitly keep track of any radix point. The only points at which the radix point position is required explicitly are in the conversions to and from floating-point, and these were obtained from the spreadsheet modelling the dataflow mentioned in section 3.3.2. The spreadsheet was also useful as a debug tool to determine the position of the radix point at different points in the pipeline during development.

As Xilinx do not offer a fixed-point package in a single IP core, solutions had to be found for each of the operations separately. They are discussed in turn below.

Multiply-Accumulate and Multiply-Add

In addition to standard arithmetic Xilinx offers Multiply-Accumulate and Multiply-Add cores for integer/fixed-point numbers in its IP catalogue. These cores are useful as the operations are common in DSP applications, and the DSP blocks are in fact multiply-adders so these cores enable more efficient usage of the hardware.

By inspecting the dataflow of our design and doing a count of the operations, we calculate that if the multiply-accumulate core is made use of wherever possible, we can make significant resource savings: it would result in replacing 13 adds and 24 multiplies with 12 multiply-accumulate cores. For multiply-add, we can replace 13 adders 13 multipliers with 13 multiply-adders and some extra shifting overhead.

Although these savings could be made, it was decided to stay with separate add and multiply cores as with the floating-point design. Use of the multiply-accumulate core would disrupt the deep, synchronous pipeline. Furthermore, the core amounts to hardware-reuse, which goes against the goal of maximum parallelism. For multiply-add, the reason was also twofold; firstly, the savings are not as significant as one would expect; 7 DSP blocks, roughly 1344 LUTs and 2352 flip-flops would be saved[4], if we ignore the additional shifting overhead. Secondly, the multiply-add core cannot be parametrised in terms of its DSP usage, possibly limiting the range of FPGAs that the design could be implemented on. Being able to control the fabric/DSP usage ratio of the design as much as possible makes the design more portable.

[4]Between 6 and 12% of the final design, see section 6.2.1.

Square root

For the task of square root the COordinate Rotational DIgital Computer (CORDIC) algorithm was used[5]. Xilinx offer a CORDIC core[60] which can be used for a number of operations, including:

- vector rotations
- vector translation between polar and rectangular coordinates
- basic trigonometry

By virtue of the fact that vector translation involves an implicit square root (the magnitude of a polar vector is equal to the square root of the sum of its rectangular coordinates), the core also offers a lightweight square root operation.

The core can perform either an integer or a fractional square root. For the fractional case, the input must be expressed as an unsigned fixed-point number with an integer width of 1 bit[60]. For our purposes, therefore, some scaling and re-interpretation of the data was required. The core datasheet provides information on how to do this, and it is discussed in appendix D. For our design the hardware requirements of this scaling amounted to two right-shifts, implemented combinatorially on the input and output of the CORDIC core at negligible cost.

As the input is an unsigned number the core does not handle the exception case of a negative input. In our design, we observe that all the inputs to square root operations are the sums of squares (see equations 3.1 to 3.5) and can never be negative. We therefore need not concern ourselves with this exception case.

Another thing to note from equations 3.1 to 3.5 is that the only need for the square root operation in our design is for the calculation of the Euclidean Norm, or the magnitude of a vector in 2D and 3D space. Although the CORDIC operator does this directly when converting from rectangular to polar coordinates, the Xilinx core works only in two dimensions. Extending the CORDIC algorithm to 3 dimensions is not straightforward, mainly due to lack of analytical expressions for the convergence of the algorithm[28], which would be necessary in order to build a core which does not disrupt the pipeline. For this reason it was decided to use the CORDIC core as a square root operator for the 3D cases. In the 2D case (equation 3.5), the calculation of the sum of squares has to be done anyway[6], so there would be no hardware saving in using CORDIC to do a vector translation instead of a square root. By performing the vector translation we can reduce the latency in performing the calculation, but recalling that this equation does not form part of the critical path (see figure 4.3), we stay with a square root to simplify the design.

[5] This algorithm was widely used in early pocket calculators, another application where the minimisation of hardware resource usage was a key goal.

[6] Recall that to calculate $x^{3/2}$ we multiply x by $x^{1/2}$ (equation 3.24).

Division

Xilinx offer an integer divider core in their IP catalogue[63] which implements hardware based on a choice of two different division algorithms: Radix-2 Non-Restoring, and High-Radix division. Radix-2 is more efficient for shorter word sizes, while the High-Radix option is more suited to longer words.

After investigations making use of this core it was decided that the division be done in floating-point. The design is thus a hybrid and not purely fixed-point, initially converting its inputs to fixed-point, then to a custom floating-point format as used in the floating-point design, and finally to single-precision in order to pass the results back to the host machine. There were three major reasons for this decision, and they are discussed in turn below.

1. The divider core has a maximum input word length of 54 bits. Unfortunately, as observed in figure 3.7 our design results in dataflows which have the largest word sizes before and after the final divide operations, and in all instances (except the Doppler-velocity partial component) this is larger than the core can accept.

2. We recall that the outputs of the system have to be converted back to floating-point anyway at some stage, in order for them to be used by the host machine. The divide is the final operation in the equations, and so is the only one that is affected by a conversion to floating-point before it. We therefore would not save on any conversion hardware by postponing the conversion until after the divide, besides the final float-float conversion converting the reduced precision format to SP. As we show in section 6.2.1, the logic footprint of this conversion is relatively small.

3. At these word sizes, the floating-point divider is in fact less resource hungry than the fixed-point version. Table 4.2 compares the floating and fixed-point division cores investigated. From it we can observe that at its maximum word size, the fixed-point core is worse than the floating-point option in almost every respect. The two fixed-point cases shown differ only in their latency parameters, which in turn affects resource usage and achievable clock speed. Case 1 achieves a high clock speed at the expense of resources and latency, while case 2 lowers latency and resource requirements but results in an acceptably low clock speed. Note also the high DSP requirements of the fixed-point cores compared to the floating-point core which uses none. The savings achieved by using the intermediate reduced-precision format can be seen to be considerable. If the floating-point cores were able to produce outputs with different precisions to their inputs, the additional float-float conversion cores could have been avoided, but even with the cost of the extra converter the reduced-precision option was more efficient than SP.

Note that at lower word sizes, requirements for the fixed-point core drop off sharply: for inputs 36 bits wide less than half the resources are used[63]. This may mean that there are potentially more

	Single Precision floating-point	Reduced Precision floating-point	fixed-point Case 1	fixed-point Case 2
Input Bit Widths	32	21	54	54
Output Bit Widths	32	21	54 + 28	54 + 28
LUT-Flip Flop pairs	1407	547	1693	1091
LUTs	780	303	1398	1009
Flip Flops	1367	492	1649	266
Clock Speed (MHz)	381	426	337	47
Latency	26	16	43	8
Block RAMs	0	0	1	1
DSP48s	0	0	17	17

Table 4.2: Comparison of the Xilinx floating-point and fixed-point dividers in terms of performance and resource requirements based on device data sheets[62, 63] and experimental synthesis. Quoted results are from builds on a Virtex 5 XC5VLX30-1 and XC5VSX35T-1 for floating- and fixed-point respectively. Both fixed-point cases implement the High-Radix algorithm, which is recommended for word sizes larger than 16. All cores are fully pipelined and produce an output at every clock cycle.

efficient designs which use more dividers, but smaller, fixed-point ones placed earlier in the pipeline, before bit growth has progressed to its maximum. Our focus in this study was simply to minimise the total dividers, and investigating this additional dimension in the solution space is left to future work.

The point at which floating-point operations become more efficient than fixed-point was found to be even earlier than division for one component in the design. In the Doppler-position partials, the word widths become too wide for the fixed-point cores two stages before the final division. Investigation showed that, as for division, at these widths the floating-point cores were also less resource hungry. In this component we thus used not only floating-point dividers, but a total of four multipliers and one adder as well.

4.4 Investigation Using Bishop Libraries

An implementation of the algorithm making use of the floating- and fixed-point VHDL library by David Bishop[22] was also investigated. The library offers all the standard arithmetic operations, and conversion and resizing functions. It also offers a choice of rounding modes and can create both signed and unsigned fixed-point hardware. The library is now part of the standard VHDL-2008 support package, but as the ISE development environment still uses VHDL-93, older versions of the library were downloaded from the project website[9] and used.

It was found that the libraries are not well suited to the XST synthesis software used, and the library website confirms this and recommends using alternative synthesis tools[9]. As a square root operator

is not provided, and XST cannot synthesis the hardware for the division operator in this library, the Xilinx cores had to be used for this as in the other fixed-point design.

A major problem with the libraries is the fact that all of the operators are combinatorial. As a result, when medium to large word sizes are used the achievable clock speed is very low when compared to implementations making use of fully pipelined cores.

It is also much more difficult to control the distribution of dedicated hardware resources such as DSP blocks, and synthesis settings had to be carefully tweaked such that the design did not require more DSP blocks than what would be feasible on hardware.

Development time using the library was much shorter than manually connecting a series of IP cores, and an implementation using the libraries with more suitable synthesis tools is recommended for future work. Having the package included in future hardware development environments would also help.

Slice Registers	20049
LUTs	70710
Block RAM / FIFOs	3
DSP blocks	120
Clock Frequency (MHz)	56.5
Latency	72

Table 4.3: Summary of the fixed-point design making use of the Bishop libraries, based on synthesis targeting a Virtex-5 xc5vlx330t FPGA. When compared with the final results of the designs using IP cores (see results section) the design was found to be inferior.

The design was a simplified version of the one using the Xilinx cores, and used a fixed word size throughout which was set to the average width of the multi-width design: an integer length of 30 and a fraction length of 5. It would thus not work in practise and served only as a brief investigation into the possibility of making use of the library. Note the amount of DSPs used - this is significantly higher than what was used by the final designs using the cores despite comparable fabric use. The clock frequency is also very low compared to the final designs. The latency is almost doubled because of the necessary use of IP cores for division and square root. Were it not for this the throughput might have been comparable to the IP cores design, even with the low clock speed, but as it stands the bishop library design is clearly inferior, at least on Xilinx hardware, and so was not investigated further. It does however exhibit a number of advantages such as quicker development time and portability across different FPGA hardware, and a closer look at a design using these libraries is suggested for future work.

4.5 Conclusions

Two hardware designs making use of Xilinx IP cores are proposed: one using reduced precision floating-point, and one being a hybrid fixed/floating-point design with multiple fixed-point formats and in which the last few operations of the pipeline are performed in reduced precision floating-point. The designs are both fully pipelined and present output at every clock cycle once the pipeline is full. Both designs are as parallel as possible, with no hardware reuse along the pipeline, and no support from a softcore processor.

Chapter 5

Implementation

We now discuss the implementation, verification and testing of the design. First, we describe the tools and methods used in implementing the design, and then discuss how it was verified by comparing its outputs to those obtained by a DP floating-point software implementation of the calculations. We then discuss the testing phase, in which a simulated system was created by interfacing C code with hardware simulations, and a portion of the design was implemented on a Virtex-4 FPGA housed on a Nallatech PCI-X coprocessor card.

5.1 Implementation

The design was iteratively coded, simulated, synthesised, and verified using ISE 12.1. The ISE development environment integrates the various tools necessary for FPGA development such as source code syntax checking, synthesis, and place-and-route to create the final bitstream to be implemented on the device. It also provides a host of analysis tools for timing, power, and resource usage analysis.

5.1.1 Coding

The design was implemented in VHDL as opposed to Verilog due to familiarity with the language. Besides the case of the investigation using the Bishop fixed-point libraries, the same designs could just as easily have been implemented in Verilog, and ISE in fact allows for designers to mix the two. Coding was mostly a relatively simple matter of instantiating cores generated with the COREGen software and wiring their data and control ports. Python scripts were made use of to automate some of the more repetitive parts of the coding process, but more high-level methods such as making use of the schematic editor feature of ISE or languages such as MyHDL could perhaps have sped up development.

5.1.2 Simulation

The Isim simulation software package, which is provided as part of the ISE development environment, was used for all hardware simulations. Isim provides the designer with a helpful GUI interface to display signal waveforms, and by visual inspection of control signals the proper functioning of the pipeline could be verified. ISE also allows the designer to view a post-synthesis Register Transfer Level (RTL) schematic of the design, in order to visually confirm that the synthesis process has generated hardware as intended at this level of abstraction. Black-box testing was performed for each component of the design; component outputs were converted from binary to floating-point decimal with the help of python scripts, and compared to expected results from the software version of the algorithm.

5.1.3 Synthesis

Xilinx Synthesis Technology (XST) software was used for hardware synthesis. Synthesis can be considered the hardware equivalent of compilation, and converts source code into a netlist which is used by the place-and-route process to map the design to the FPGA fabric.

The Xilinx IP cores which make up the design are generated as "black box" components, which are pre-synthesised. When the HDL compiler comes across a component declaration with a 'black box' attribute, it looks for the pre-generated netlist file instead of VHDL source code, and uses the netlist directly for synthesis without having to compile the source first. A warning is displayed, but otherwise the compiler proceeds on the assumption that the component is error-free when a component is synthesised in this way. This results in greatly reduced synthesis times for the design and accelerated development.

There is one major downside to making use of pre-synthesised black box components, however: it means that possible trims and hardware savings unique to our design are not taken advantage of in the place-and-route process. For example, in their investigation of a custom euclidean norm operator for FPGAs, [31] reported that significant resource savings were achieved simply by resynthesising the arrangement of floating-point adders, multipliers, and square root hardware from the source code instead of implementing black boxes. The synthesis software was able to automatically eliminate unnecessary hardware such as that required for negative results from squarers. As the proprietary IP provided by Xilinx is exclusively black-box, our design unfortunately misses out on these potential benefits. With access to the source code the entire design could be more efficient in terms of resource usage.

Synthesis/toolflow options

An FPGA design does not end with HDL description; synthesis, mapping, and place-and-route are complex processes which should be considered closely for an efficient design. Once the design

was completed, synthesis and mapping options were experimented with, setting them to optimise for area or speed, and noting the effect of enabling options such as *register duplication* (which replicates registers to help reduce fanout) and *retiming* (which rearranges registers to improve timing results). Experimenting with these parameters is something of a trial-and-error exercise, as the results are not always as expected. For example, builds set to optimise for speed resulted in even lower clock rates for our design. Although an extensive search of build-strategies was not carried out, it was found from the tests performed that the default settings produced satisfactory results, with a good balance between resource requirements and timing performance.

5.2 Verification

Hardware verification was performed using hand written VHDL test benches and Isim. A few output values were verified manually, and python scripting was then used to perform a more realistic test using data obtained from runs of the C implementation of the algorithm.

Using the software version of the algorithm, input and output values of the code hotspot were saved to text files to an accuracy of 50 significant digits for every simulation in the dataset. In order to reduce this data to a practical size, only every tenth input-output pair was used for verification, as neighbouring pairs tended to be very similar. These were converted to text versions of their binary representation in python (script included in appendix C), and this was then passed to Isim to be used as inputs to the test bench. The Isim behavioural simulation produced another text file containing binary representations of the hardware outputs, and these were read by the python script, converted to floating-point numbers, and compared with the results of performing the calculations in DP floating-point in python. The process is shown in figure 5.1.

It was not determined exactly how close to the software results the hardware would have to be at this point in the code for proper functioning of the complete algorithm, and the comparisons were only done on an iteration-by-iteration basis i.e., each iteration was verified in isolation, without taking into account the fact that in practise the iteration inputs depend on their own previous outputs. Nevertheless, errors of less than 1% for all outputs was achieved on average, and this was assumed to be sufficient to proceed.

5.2.1 Implementation Experiment on Coprocessor Card

The concept of accelerating computation with FPGA based coprocessors is not a novel one, and a variety of coprocessor platforms have been commercially available for years. One such product is the H101-PCIXM Accelerator card from Nallatech[14]. It is a PCI-X expansion card housing a Xilinx Virtex-4 FPGA as well as external memory and off-card serial I/O. The specifications of the card and the FPGA it houses are given in table 5.1b and 5.1a respectively.

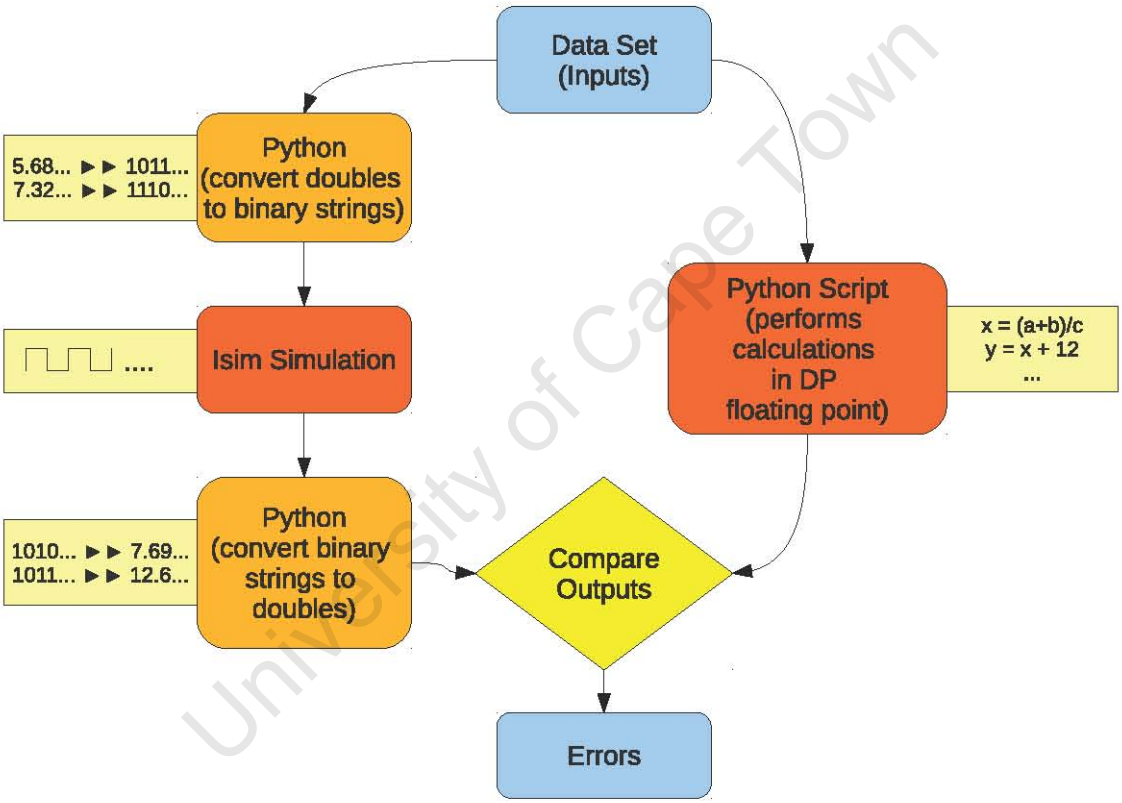


Figure 5.1: Block diagram of the verification process.

A section of the floating-point design was tested by running it on the card. Although a complete system with communication between the C implementation of the algorithm and the card was not implemented , implementing it on physical hardware served as a final verification of the design. The Virtex-4 series of FPGAs was introduced in 2004, and is thus starting to show its age when compared to the more recent Virtex-5 and Virtex-6 series. As it was not big enough for 8 instances of the entire processing core, a subset of it was implemented, namely the Doppler-velocity partials calculations. For this subsection 8 instances could be implemented in parallel for a slightly more realistic proof of concept.

Virtex-4 LX100	
Slices	49152
Max. Distributed RAM bits	786432
Total Block RAM (kbits)	4320
Max Select I/O	960
DSP blocks	96

(a)

Nallatech H101	
Form Factor	64-bit/133MHz PCI-X
Host Bandwidth	Up to 400MB/s
FPGA	Xilinx Virtex-4 LX100-10
SDRAM Memory	0.5GB DDR2 SDRAM
SRAM Memory	16MB DDR-II SRAM
Serial I/O	4x 2.5Gbps via PCI backplate

(b)

Table 5.1: specifications for the Nallatech H101 Accelerator card and the Xilinx Virtex-4 LX100 FPGA it houses.

Nallatech also offer an FPGA development environment known as the DIME toolchain. This consists primarily of DIME-C, a C-to-VHDL compiler, and DIMEtalk, GUI-based a design tool which abstracts away much of the hardware design details. DIMEtalk was made use of for programming the FPGA. A project was created in DIMETalk with nodes for a clock controller, PCI interface core, and FIFOs for each input and output stream. The ISE project was then imported and connected as a 'user VHDL component'. In order to do this the project was first synthesised as a black box, and this was then imported to the DIMETalk framework, treating the design as a single core. FIFO primitives were instantiated in DIMETalk as opposed to the other I/O options (BRAMs or the on-board SDRAM) as this improved latency and simplified control logic. Although the GUI interface does simplify development, ports still have to be connected manually and this can involve tedious, repetitive work. The DIMETalk network is shown in figure 5.2 below.

It was found that the major limitation of the Virtex-4 FPGA for our design was the number of DSP blocks. For our design the number of multipliers exceeded the number of DSP blocks and they had to be used sparingly, with only 12 DSPs available per PE. The operators to be allocated a DSP block were chosen such that the savings in latency would shorten the critical path and thus benefit the design as a whole. All 96 of the DSP blocks were used by multipliers.

After a comparison with results from performing the calculations in DP floating-point in software, The design was found to perform as expected in terms of accuracy, with all outputs within 1% of their software equivalents.

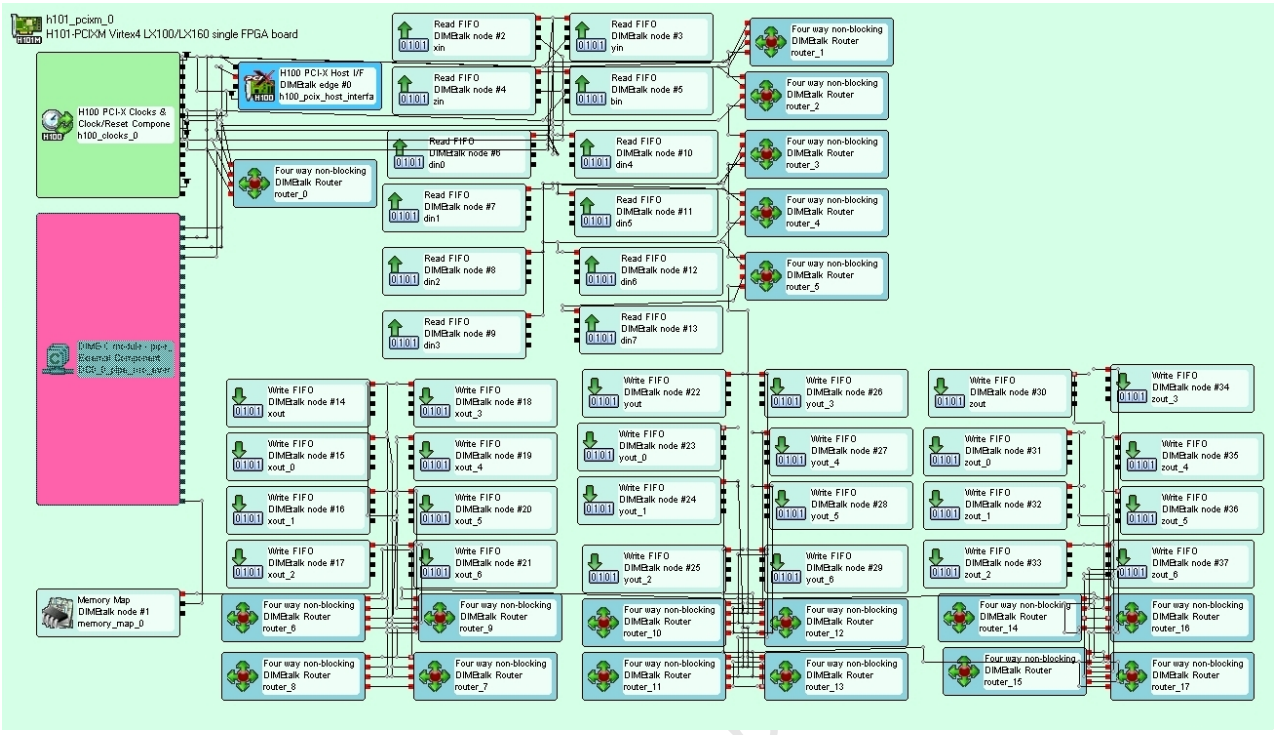


Figure 5.2: DIMETalk network created to facilitate programming the Nallatech coprocessor card. For simplicity much of the wiring has been removed - in the actual design every port of the GN design (the pink component at left) had to be manually connected to a FIFO block.

5.3 Testing

Testing is distinct from verification in that where verification checks whether what has been built is what was originally intended, testing checks whether what has been built performs properly. Our verification steps confirmed that the equations were being performed correctly in isolation, and our testing stage checks if the integrated system works correctly, with data communicated between the host and device.

A simulated system was tested by making calls to the same Isim simulation described in section 5.2 from within the C code for every iteration of the algorithm, with data communicated via text files as in the verification stage. Figure 5.3 displays the system.

This provided a realistic simulation with a bit-true representation of the custom number system, although of course no information about potential speedup was acquired in this way as the Isim simulations are much slower than the actual device would be. The system was tested using all the data from the dataset, and in all cases was found to perform just as well as the original software version in terms of precision. The results are presented in chapter 6.

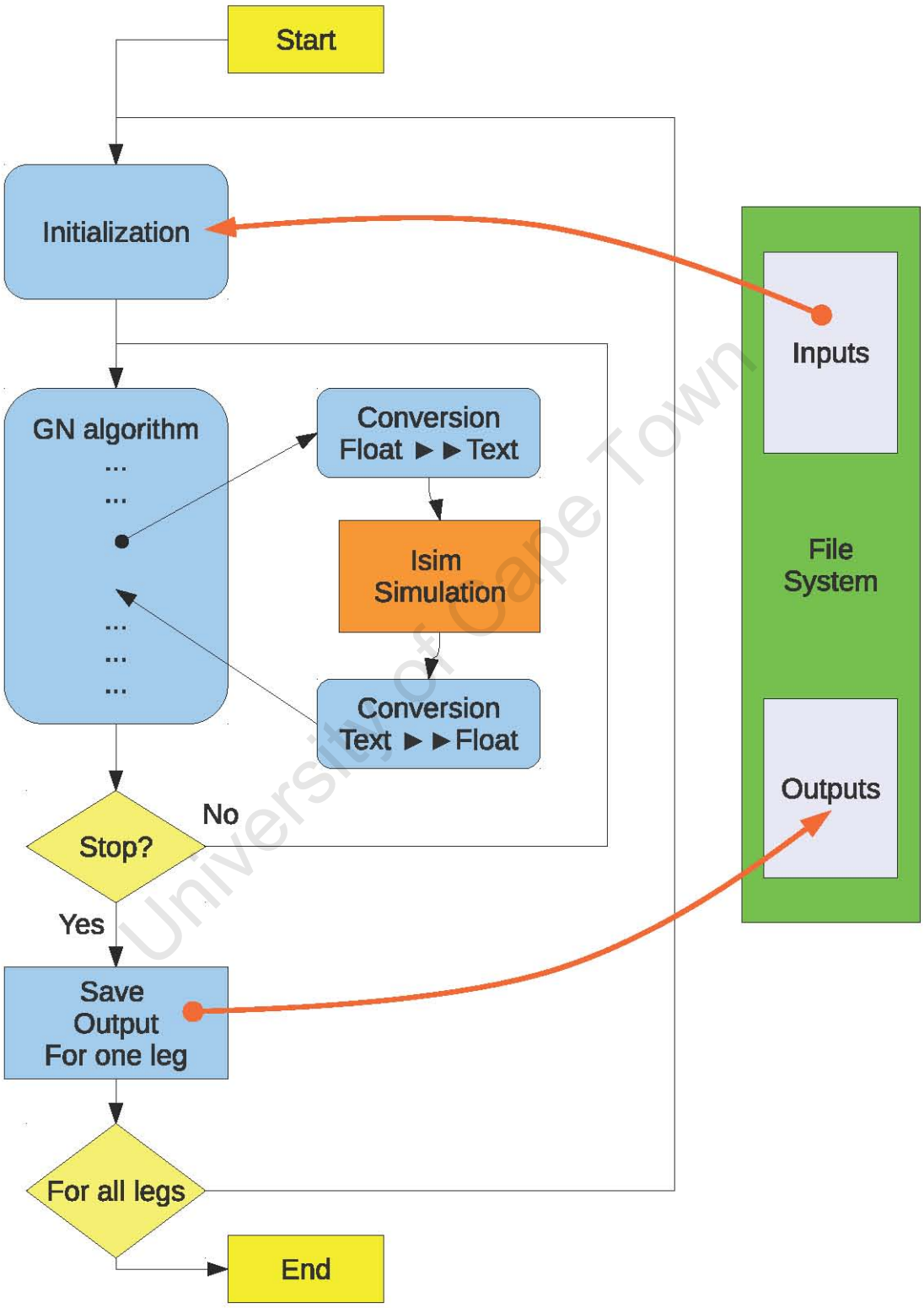


Figure 5.3: Block diagram of the testing process. The original software version of the algorithm was edited such that the calculations were performed by an Isim simulation for every iteration.

5.4 Conclusions

Both designs were successfully implemented in HDL targeting FPGAs, with verification showing that they perform almost identically to the software version in terms of accuracy. This verifies the algorithm profiling described in section 3.3. The simulated system used for testing has been described, and a section of the design was also programmed on to a Virtex-4 FPGA on a Nallatech coprocessor card, and its functionality was verified.

Chapter 6

Results

We now present the results of implementing the design. First we discuss the results in terms of the filter accuracy, and show that both the reduced-precision floating-point and the fixed-point design perform just as well as the original software version. We then discuss the final hardware designs in terms of four metrics: resource usage, achievable clock speed, pipeline latency, and power consumption. Finally we present a theoretical speedup, based on the latencies and achievable clock speeds of the designs and on the latencies of available host-device communication options.

6.1 Accuracy Results

It was found that the performance in terms of accuracy of the reduced precision implementation was very close to that of the original double-precision code. Figure 6.1 compares the accuracies of the final filter outputs for the fixed-point, floating-point, and original software implementations, as well as the average number of iterations to convergence. A closer look at the data is included in appendix E. It can be seen that the accuracies are on average very similar for all outputs and all implementations. There is however a slight increase in the average iterations till convergence in the fixed-point implementation.

6.2 FPGA Hardware

We discuss the results of implementing the hardware designs in terms of four metrics: resource usage, achievable clock speed, pipeline latency, and power consumption.

6.2.1 Resource Usage

A comparison of resource requirements for the two designs is surprising in that the fixed-point version requires more resources than the floating-point version. It was found that the major reason

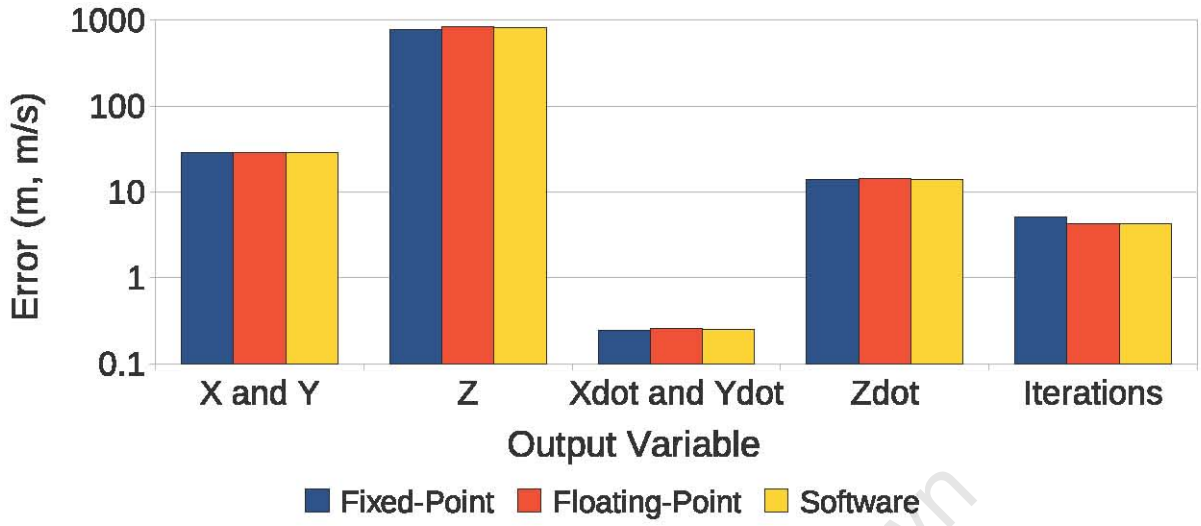


Figure 6.1: Average error comparison for output variables, and a comparison of the average iterations to convergence over the entire data set.

for this is that the conversion from floating-point to fixed-point requires much more hardware than conversion from one floating-point format to another. This makes sense intuitively - in our floating-point design, because the exponent width remains 8 bits, the conversion to our floating-point format simply involves discarding a few of the least significant mantissa bits, and using the conversion cores at all was perhaps unnecessary.

In addition to this, on the fixed-point side there is the fact that more than one fixed-floating conversion is performed for every output. Recall from section 4.3 that the final operations of the design are done in floating-point, and thus both inputs to the cores have to be converted. There is also the final additional conversion back to single precision from the custom format. For the fixed-point design the cost of conversion hardware amounted to 7920 slice registers and 5978 LUTs, whereas the floating-point design used only 1240 slice registers and 1003 LUTs (the conversion cores do not make use of any dedicated hardware such as DSP blocks). Results from synthesis showing the difference in conversion resources is given in table 6.1

The number of FIFO primitives used on the FPGA is much lower than the number of instantiated cores (of which there are more than 20). It was initially assumed that this meant not all instantiated cores were making use of the FIFO hardware, but further investigation revealed that the amount of logic (in addition to the FIFO hardware) used for shifting is much less than what one would expect if shifters were being implemented in logic. This suggests that the synthesiser is able to share FIFO hardware between multiple FIFO cores. The FIFO primitives are 1024 bits wide and 512 bits deep - none of the instantiated cores ever have to use even half of this.

When synthesising only the hardware for the arithmetic cores themselves, the fixed-point design was

Synthesis	Floating-Point Minimum DSP	Floating-Point Maximum DSP	Total Available		
			LX330T	LX110T	SX95T
Slice Registers	23856	16576	207360	69120	58880
LUTs	19576	11736	207369	69120	58880
Slice Registers (conversion)	1240	1240	-	-	-
LUTs (conversion)	1003	1003	-	-	-
BRAM / FIFOs	9	9	324	148	244
DSP blocks	0	80	192	64	640
Clock (MHz)	262.8	262.5	-	-	-
Latency	115	118	-	-	-

(a)

Synthesis	Fixed-Point Minimum DSP	Fixed-Point Maximum DSP	Total Available		
			LX330T	LX110T	SX95T
Slice Registers	37470	20329	207360	69120	58880
LUTs	31774	14836	207369	69120	58880
Slice Registers (conversion)	7920	7920	-	-	-
LUTs (conversion)	5978	5978	-	-	-
BRAM / FIFOs	6	4	324	148	244
DSP blocks	2	83	192	64	640
Clock (MHz)	255.6	272.778	-	-	-
Latency	118	119	-	-	-

(b)

Table 6.1: Comparison of resource usage for one PE. Results are from synthesis estimations using default synthesis settings, targeting a Virtex-5 xc5vlx330t FPGA. The fixed-point design requires at least two DSP blocks because some of its multipliers are set to "multiply by constant" mode, which cannot be set to use only logic.

approximately 20% smaller than floating-point with full DSP usage. When no DSPs were used, the design was found to be 96% bigger. This is because the resource requirements of the fixed-point cores rises steeply with input size when they are not allowed to make use of embedded multipliers, and the variables in our data path are almost all wider than 12 bits, which is the largest input size quoted in the "Resource Utilisation" section of the fixed-point multiplier core datasheet when no DSP blocks are used[64]. Perhaps this implies that for inputs wider than 12 bits, DSP usage is recommended.

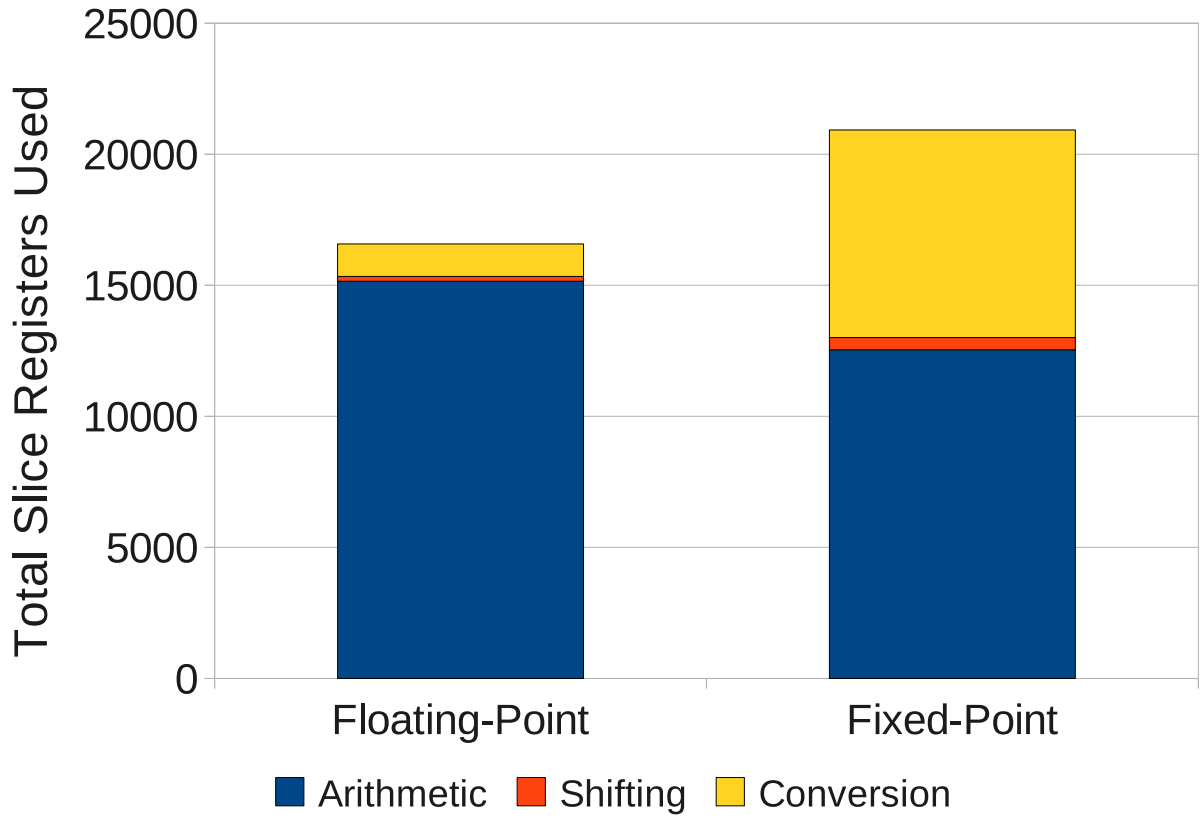


Figure 6.2: Distribution of resources in the fixed-point and floating-point designs with maximum DSP usage. Although the fixed-point design is slightly smaller in terms of pure arithmetic usage, the amount of resources needed for conversion make it a larger design in total. The values for shifting show only the logic used and do not reflect the amount of dedicated FIFO hardware that was also made use of by each design. The higher resource requirement for shifting in the fixed-point design can be attributed to the fact that the word sizes were larger than the floating-point ones for most of the pipeline.

As synthesis results are only estimates, for actual results a full place-and-route should be performed. The actual resource usage and timing results are shown below in table 6.2. They show similar resource usage values, but a significant drop in achievable clock speed for all designs.

We note some difficulties in the results: with full DSP usage, only two PEs would fit on the LX330T, and because of the low slice count on the SX95T not more than two would be possible on this

Post Place-and-Route	Floating-Point Minimum DSP	Floating-Point Maximum DSP	Total Available		
			LX330T	LX110T	SX95T
Slices	8972	6597	51840	17280	14720
Slice LUTs	19425	11763	207360	69120	58880
Slice Flip Flops/Registers	23466	16403	207369	69120	58880
Slice LUT-Flip Flop pairs	25415	18122	207369	69120	58880
FIFOs	23	23	324	148	244
Block RAM / FIFOs	9	9	324	148	244
DSP blocks	0	80	192	64	640
Clock Frequency (MHz)	177.399	162.999	-	-	-
Latency	115	118	-	-	-

(a)

Post Place-and-Route	Fixed-Point Minimum DSP	Fixed-Point Maximum DSP	Total Available		
			LX330T	LX110T	SX95T
Slices	12760	7972	51840	17280	14720
Slice LUTs	32258	15365	207360	69120	58880
Slice Flip Flops/Registers	37269	20025	207369	69120	58880
Slice LUT-Flip Flop pairs	41732	23594	207369	69120	58880
FIFOs	24	24	324	148	244
Block RAM / FIFOs	6	4	324	148	244
DSP blocks	2	83	192	64	640
Clock Frequency (MHz)	168.719	175.778	-	-	-
Latency	118	119	-	-	-

(b)

Table 6.2: Comparison of resource usage for one PE. Results are from post place-and-route, using default synthesis settings, targeting a Virtex 5 xc5vlx330t FPGA. The only constraints set were those for timing, aiming for a clock speed of 200MHz, which was never achieved.

device either. For a practical implementation, either an FPGA higher up in the SX range should be used, or some of the cores would have to be implemented purely with logic, with usage results somewhere between the given values for maximum and minimum DSP use. In this case, the available DSPs should be placed on the critical path of the pipeline, so that their reduced latency benefits the design as a whole, as in the Nallatech coprocessor card investigation of section 5.2.1. Note also that problems to do with available real estate on the FPGA will quickly become less significant as new generations of the devices are developed.

Resource/precision tradeoff

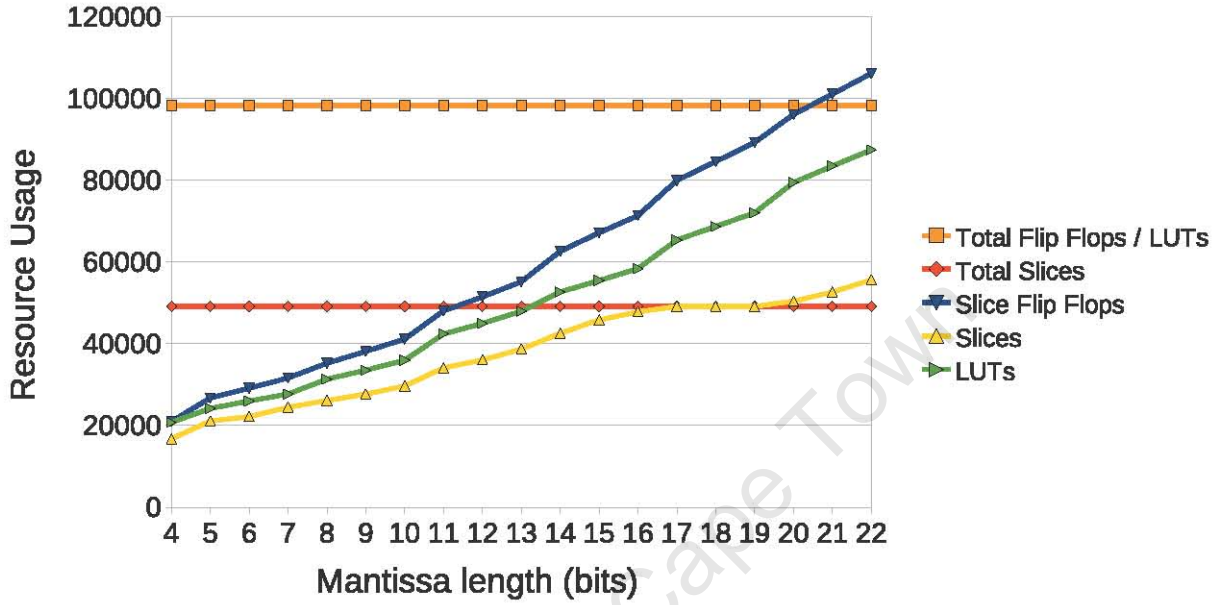
In order to investigate the relationship between precision and FPGA resource requirements, a portion of the design was configured and synthesised at various exponent and mantissa widths, with the help of python scripting (see appendix C). Although the floating-point core datasheet does indicate a relatively linear increase in resource usage with increased precision, this does not necessarily imply that the design as a whole will behave this way as there are other factors such as FIFO shifter widths and latency changes. The hardware used in these measurements was a subset of the complete design, consisting of 8 parallel instances of the Doppler-velocity partials component, but this component can be considered representative of the entire design. The results are shown in figure 6.3.

Scaling

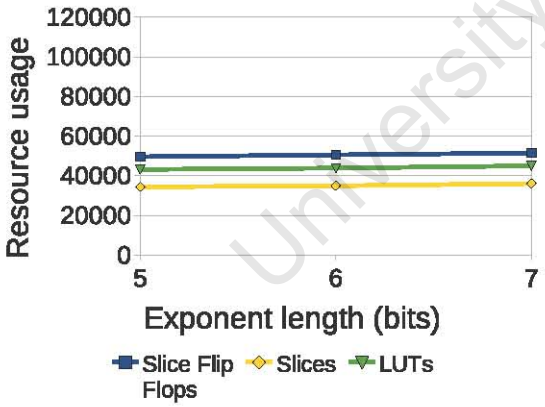
The scaling of resource requirements with the addition of receiver processing units was investigated. The resource usage will not necessarily increase linearly, as subtleties such as the course grained nature of DSP blocks and the sharing of FIFOs may come into play. In our design, however, it was found that there is a linear increase in requirements with number of processing units. Unfortunately, this also includes IO pin requirements; output bandwidth scales linearly with the number of processors in our design, and the output would probably have to be serialised somehow in order to reduce the number of IO pins used for a practical design. Although the logic resources of a large Virtex-5 could fit about ten PEs, there will only be enough I/O pins for one or two.

6.2.2 Achievable Clock Speed

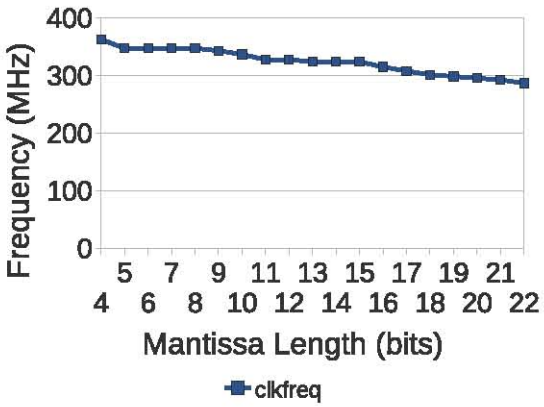
With respect to clock speed the designs fare equally well, each estimating an achievable clock speed speed of more than 150MHz. The slowest asynchronous signal in the design (which determines the achievable clock speed) was always found to reside within one of the arithmetic cores, indicating that more efficient control logic or placement of the cores would not speed up the clock.



(a) Resource usage versus mantissa length. The usage rises fairly linearly with word size. Note the slice usage stopping at its maximum for as long as possible - this is due to the place and route process working harder to fit the design.



(b) Comparison of exponent widths, with the y-axis range set to that of figure A in order to show the relatively negligible effect of exponent size on hardware resources.



(c) Achievable clock speed versus mantissa length. The clock speed drops fairly linearly with word size.

Figure 6.3: Synthesis results of a subset of the design, with 8 parallel instances of the component which calculates Doppler partials with respect to velocity. Resource usage increases linearly with increasing mantissa length, and clock speed decreases linearly as precision increases.

6.2.3 Latency

The reason for the long latency in the fixed-point design is the CORDIC square root core. At the necessary word widths at this point in the data path, the core has a very long latency of 41 cycles. This core forms part of the critical path of the design and thus directly affects the latency of the design as a whole. Although requiring more hardware resources, the floating-point square root core has a latency of only 17 cycles. Were it not for the very long latency of the square root operation, the fixed-point design would have a latency much shorter than that of the floating-point, as for all other cores used the latency is significantly less in the fixed-point case.

6.2.4 Power

According to the Xilinx XPower analyser, a tool included as part of the ISE design suite for analysing the power consumption of FPGA designs, the floating-point design making full use of DSP hardware will consume approximately 3.295W. As the power consumption depends mostly on clock speed (a fact which was also confirmed with the tool), and all the designs have similar clock speed they all consume roughly the same amount of power. This is much less than the power dissipated by a high performance CPU or GPU, so in this respect our design does well in taking advantage of a strength of FPGAs.

6.3 Projected Speedup

As a fully integrated system was not developed, we can only provide theoretical speedup results.

Our first task is to measure the time taken for the code hotspot in the unaccelerated code. According to timing results of the software implementation running on an Intel Xeon 3GHz processor, the function containing the arithmetic ported to hardware takes $\mathbf{0.62}\mu s$ on average for each time it is called. For a single iteration of the algorithm, this function is called for every receiver and every sample in a leg. The total time taken for one iteration (which represents the amount of work done by the hardware core) thus amounts to $\mathbf{399.57}\mu s$ for a single-threaded design[1].

From the post place-and-route results of the best design (floating-point with maximum DSP usage), to be safe let us assume that our implemented FPGA hardware will have a clock frequency of 150MHz, and have a latency (excluding any of-chip communications) of 118 clock cycles. To process 80 inputs for one iteration of the algorithm would then take 198 clock cycles for the fully pipelined design. Thus if the calculations for each receiver are performed in parallel, the total processing time for one iteration is $\mathbf{1.32}\mu s$. The theoretical speedup purely for computation is thus 399.57/1.32 = **302.7**.

[1] These timing results were obtained by making calls to the system clock using functions from the C Standard Library and averaging over multiple runs of the code.

Of course, communication to and from the FPGA should be an important consideration of the final design, and any speedup calculations must take this into account. Although no physical system was built, we present some theoretical projections based on quoted communication speeds.

The Nalletech cards used in section 5.2.1 quote a maximum host bandwidth of **400MB/s**. For every iteration of the algorithm, we need to transport 80 samples for each of 11 inputs, in single-precision floating-point which is 4 bytes wide. This gives us a total of 3520 bytes on to the FPGA. The results to be passed back to the host consist of 80 samples for each of 9 outputs, but for each receiver (let us assume 8), also in single-precision. This amounts to 23040 bytes off of the FPGA, and a total communication load of **26560 bytes** per iteration. If we communicate at max bandwidth this means communication will take **66.4**μs for each iteration. Taking this into account then, we obtain a more realistic speedup value of 399.57/65.72 = **6.08**. Interestingly, the actual computation time has become a negligible factor in the speedup and it depends almost wholly on communication speed.

The cards communicate via a PCI-X bus, which is far from the fastest communication protocol available today. If we consider PCI express, we could potentially have the FPGA communicating over 16 lanes at 500MB/s per lane[2], resulting in a theoretical maximum bandwidth of **8GB/s**. At this speed we would achieve a speedup of **86.12**. Of course this bandwidth would probably not quite be achievable in practise, but achieving even half of it would result in a significant speedup.

These results are still simplistic in that they assume that the FPGA can take in all the inputs for each sample in one clock cycle, and similarly present all outputs for a sample at once. As the latency of the pipeline is longer than 80 cycles it will never have to do input and output at the same time, but even so there are not enough IO pins for more than one or two PEs on even the largest Virtex-5 FPGAs if communication is to be fully parallel. There will thus have to be some sharing of pins resulting in increased latency on the FPGA, but as we have seen, latency on the device-side is a negligible issue for our design.

6.3.1 Scaling

Speedup

As including more processing cores on the FPGA will not affect the processing latency, and should not affect achievable clock speed significantly, the only obstacle preventing superlinear speedup (in theory) is increased bandwidth. Based on the assumptions made in the previous section, figure 6.4 shows how the expected speedup for various communication bandwidths scales with increased parallelism.

[2]For PCI-e v2.x. The final specifications for v3.0 are delayed until 2011, but is it expected to be capable of up to 1GB/s per lane[6].

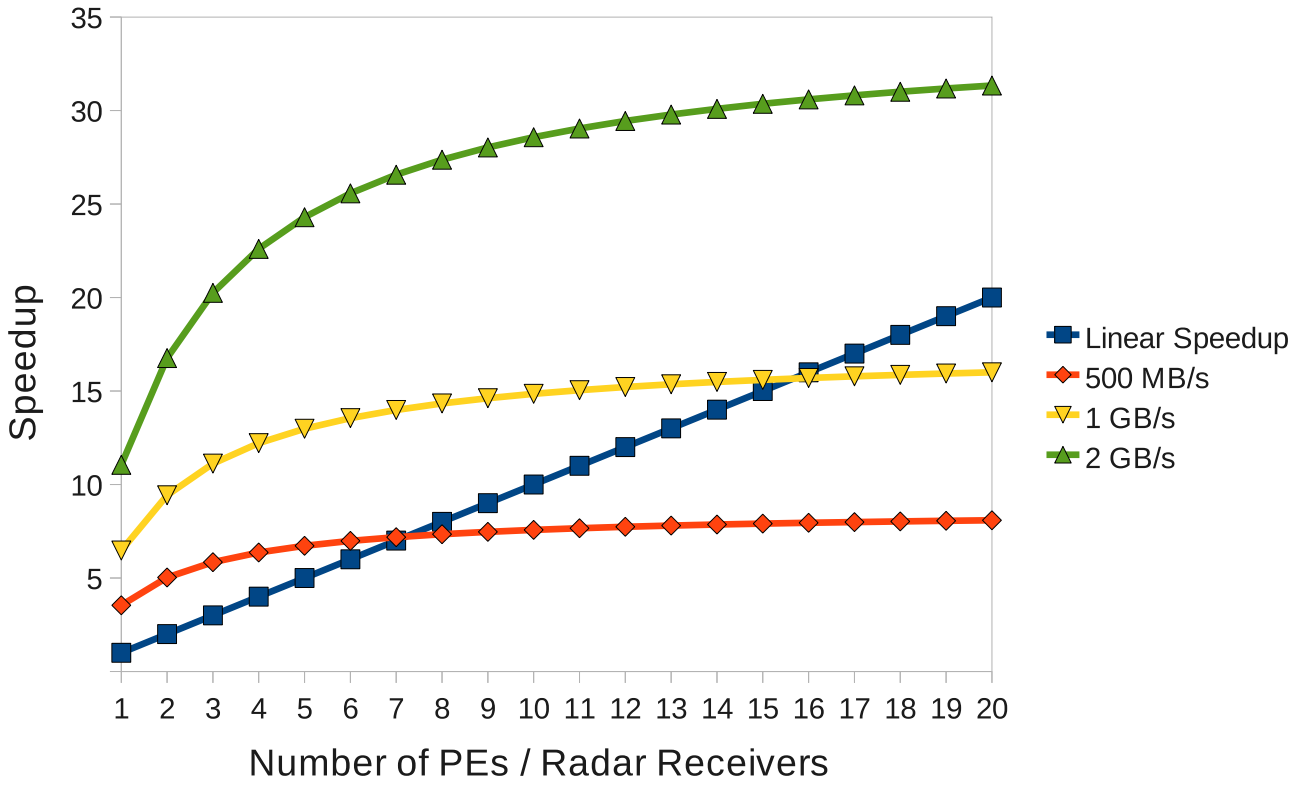


Figure 6.4: theoretical scaling of speedup with number of parallel PEs. Due to the linear increase in bandwidth requirements with increasing PEs, speedup reaches an asymptotic limit, as communication time dominates the total latency.

Chapter 7

Conclusions

Two hardware designs for implementation of the Gauss-Newton algorithm on an FPGA coprocessor system have been developed, verified and tested. Simulation results show that the system is just as accurate as a software implementation, and the projected speedup results look very promising.

Results of profiling the software version of the algorithm confirm the conclusions of previous work, namely that the performance bottleneck of the algorithm is the calculation of the partial derivatives that make up the sensitivity matrix. Although this section of code sits within an unrollable loop, there is significant parallelism that can be taken advantage of, and by considering it a streaming application we were able to create a deep custom pipeline for high throughput.

In addition, software simulation revealed the algorithm does not need to use full double-precision floating-point numbers in this bottleneck portion of the code, and can tolerate a surprising reduction in dynamic range and precision. Bit-true simulations confirmed that a floating-point number system with an 8-bit exponent and a 12-bit mantissa is sufficient. Using fixed-point numbers, a custom datapath with multiple word sizes which stay less than 128 bits wide would work, though it was found that a hybrid system which switched to floating-point towards the end of the pipeline was more efficient. Neither of these reduced-precision formats affected the final accuracy of the algorithm adversely, and the only noticeable difference was a slight increase in the average iterations to convergence with the fixed-point design. This should not affect the final speedup greatly, and adding a few extra bits of precision would likely eliminate this small discrepancy.

Surprisingly, the reduced precision floating-point design required less FPGA resources than the fixed-point design. This was because the conversion from single precision to fixed-point used much more logic than the conversion between floating-point formats. The most expensive operators (the dividers) were also implemented in floating-point in this design, further reducing the potential savings of the fixed-point design.

According to theoretical projections, the performance bottleneck has been accelerated to such an extent that its processing time is negligible when compared to host-device communication time. Unfortunately, the bottleneck has a low communication to data ratio, which also scales linearly with the

number of parallel PEs. Even so, we predict that in an eight-receiver system with a communication speed of 400MB/s between host machine and device, a speedup of 6.08 can be achieved on the code hotspot. With a bandwidth of 8GB/s (the theoretical maximum of PCI-e v3.0), we could expect a very impressive speedup of 86.12.

These speedups are however only relative to the code hotspot, and when considering at the entire algorithm, Amdahl's law comes into effect and we are left with a much more modest speedup result. This work has however shown that FPGAs can be successfully used to speed up even floating-point based problems which are not embarrassingly parallel. With rapidly increasing logic densities and dedicated hardware resources on these devices, a more significant speedup will soon be easy to achieve on the algorithm as a whole.

7.1 Comments on FPGA Development

In terms of development time, virtually any other hardware platform would be much faster to develop with. Joe Milburn estimated a learning curve of approximately 3 weeks for the ClearSpeed[52], and using GPUs would also be much faster as they make use of a subset of the well known C language. Although tools such as ISE are a great help, they are not as mature as their equivalents for software development. Designers applying FPGAs to HPC will also most likely come from a software development background, and may find hardware design surprisingly different and challenging compared with what they are used to.

7.2 Future Work

The main recommendation for future work is that the study presented in this dissertation be extended by porting more of the algorithm to hardware, possibly targeting a bigger FPGA. Performing more of the work in hardware will result in the impressive speedups on those sections being more significant to the algorithm as a whole, and would hopefully improve the communication to computation ratio, which we have seen is very important in this application. With more computation offloaded, the ratio of hardware used for number conversion to that used for actual computation should also improve, making the system more efficient and possibly making fixed-point the better option.

Future work should preferably make use of abstraction-layer tools such as MATLAB Simulink. The VHDL coding for this study was time consuming and repetitive, and abstraction tools would save on both development time and time taken familiarising oneself with the development environment. Development time for the investigations using the VHDL library was much shorter than manually connecting the IP cores, and an investigation using these libraries with more suitable synthesis tools is also recommended. Having the package included natively in future hardware development environments would also help.

The work presented here could also be improved upon in a number of other ways:

- The effects of different rounding modes for the custom number formats on resource usage and final accuracy have not been investigated.
- Not making use of multiply-adders and multiply-accumulators in favour of a more parametrisable fixed-point design was perhaps a mistake, as they ability to create highly specialised designs is the main strength of reconfigurable computing. These operators would probably have saved substantial logic resources, and should at least be considered for future designs.
- Although minimising the use of more expensive operators such as square root and division at all costs seems a good idea intuitively, there could perhaps be more efficient designs which use more of these operators, as resource requirements are not only dependent on the operator type but also on input word widths and DSP usage.

An interesting avenue of investigation could perhaps be the application of genetic algorithms or similar solution-space searching optimisation tools for fixed-point FPGA designs. The solution space has many dimensions: The arrangement of equations, which determines how many operators of each kind are used, and where they sit in the pipeline. Position in the pipeline then determines the widths of input variables, which in turn affects latency and resource usage for each core. There is then the problem of how limited DSP blocks should be allocated. In this study they were placed purely to reduce latency, but perhaps there exist other solutions that take into account other factors such as resource savings and achievable clock speed. Another dimension is the effect of the design on the algorithm's average iterations to convergence - there may be solutions where this is much higher than the pure-software average, but other advantages outweigh this cost.

Finally, we recommend that future work also looks at improving the C code and making use of multiple threads, so that speedup comparisons are more fair.

Appendix A

Precision Profiling Functions

Simple functions were created in C to simulate running portions of the code at reduced precision. They are presented below.

A.1 Floating Point

For the floating point simulations, the last few bits of the single-precision floating point number are set to zero and this rounded result is returned. The operation was applied to the result of every arithmetic operation in the section of code to be offloaded to hardware.

```
1  float round_off(float input, int x)
2  {
3          short bits_to_chop = 23 - x;
4          if (bits_to_chop < 1)
5          {
6                  return input;
7          }
8
9          union flint
10         {
11                 float floa;
12                 int inte;
13         }bits;
14         bits.floa = input;
15
16         int mask = 0xffffffff << bits_to_chop;
17         //printf("mask after invert: %d\n", mask);
```

```
18
19          bits.inte = bits.inte & mask;
20
21          return bits.floa;
22  }
```

A.2 Fixed Point

The functions used for the fixed-point simulations were included as a header file. They consist of:

- A function which initialises global variables, to be called at the start of the algorithm
- The "conversion" function, which returns the closest number to its input that could be represented by a fixed-point number of dimensions set by the global variables. If the input is too great (an overflow), it returns the largest number that can be represented, and similarly returns the most negative number in the case of underflow. This operation was applied to the outputs of every arithmetic operation in the section of code to be offloaded to hardware.
- "fixed square-root" and "fixed pow" functions which simply apply the conversion function to the result of a sqrt or pow operation
- A function representing the zero-checking that was applied in hardware to the denominators of divisions. It replaces input too small to be represented by the chosen fixed-point format with the smallest representable number, instead of zero which would otherwise be the case.

```
1  #ifndef FIXED_H
2  #define FIXED_H
3
4  #include <math.h>
5
6
7  int INTBITS;
8  int FRACBITS;
9  int TOTALLENGTH;
10 //const int totalLength = 20;
11
12 int oFloFlag;
13
14 int fixedPointIterator;
15
16 double fixedPointValue;
17 double fixedPointValueFrac;
18 double max_possible;
19 double min_possible;
20 double min_pos_possible;
21
22 //       printf("max number = %f\n", max_possible);
23
24 //       printf("min number = %f\n", min_possible);
25
```

```
26 void fixedInit(int INT, int FRAC)
27 {
28     INTBITS = (INT);
29     FRACBITS = (FRAC);
30     TOTALLENGTH = INTBITS + FRACBITS;
31     max_possible = (pow(2,TOTALLENGTH - 1) - 1)/pow(2,FRACBITS);
32     min_possible = -1*(pow(2,TOTALLENGTH - 1)/pow(2,FRACBITS));
33     min_pos_possible = 1/pow(2,FRACBITS);
34     printf("\nint: %d frac: %d length: %d\n",\
35             INTBITS, FRACBITS, TOTALLENGTH);
36     printf("max: %e min: %e min_pos: %e\n",\
37             max_possible, min_possible, min_pos_possible);
38 }
39
40 double convToFixed(double val)
41 {
42
43     //check limits
44     if(val > max_possible)
45     {
46         //printf("OVERFLOW! %f > %f!\n", val, max_possible);
47         oFloFlag += 1;
48         //exit(1);
49         return max_possible;
50     }
51     else if(val < min_possible)
52     {
53         //printf("UNDERFLOW! %f < %f!\n", val, min_possible);
54         oFloFlag += 1;
55         //exit(1);
56         return min_possible;
57     }
58     else
59     {
60         //save frac bit and at the same time
61         //assign int part to fixedPointValue
62         fixedPointValueFrac = modf(val, &fixedPointValue);
63         //int part is now stored in fixedPointValue
64         //rac part is now stored in fixedPointValueFrac
65
66         fixedPointValueFrac -= fmod(fixedPointValueFrac, pow(2, -1*FRACBITS));
67         //blunted fraction part is now stored in fixedPointValueFrac
68
69         fixedPointValue += fixedPointValueFrac;
70
71         //check for errors - number can't be too precise,
```

```
72          //too big, or too small
73          if(fabs(val - fixedPointValue)>min_pos_possible \
74              || fixedPointValue > max_possible \
75              || fixedPointValue < min_possible)
76          {
77              printf("changed %f to %f\n", val, fixedPointValue);
78              printf("something went wrong...\n");
79              exit(1);
80          }
81
82          return fixedPointValue;
83      }
84  }
85
86  double fixed_sqrt(double val)
87  {
88      if(convToFixed(sqrt(val)) != convToFixed(sqrt(val)))
89      {
90          printf("sqrt made a NaN (input was %f)\n", val);
91          exit(1);
92      }
93      return convToFixed(sqrt(val));
94  }
95
96  double fixed_pow(double arg1, double arg2)
97  {
98      if(convToFixed(pow(arg1,arg2)) != convToFixed(pow(arg1,arg2)))
99      {
100         printf("pow made a NaN (input was %f ^ %f)\n", arg1, arg2);
101         exit(1);
102     }
103     return convToFixed(pow(arg1,arg2));
104 }
105
106 void CantBeZero(double *val)
107 {
108     if ( fabs(*val) < min_pos_possible )
109     {
110         if(*val >= 0)
111             *val = min_pos_possible;
112         else
113             *val = -1*min_pos_possible;
114     }
115 }
116
117 #endif
```

Appendix B

Detailed Fixed-Point Precision Profiling Results

This section presents fixed-point precision profiling results over the entire dataset.

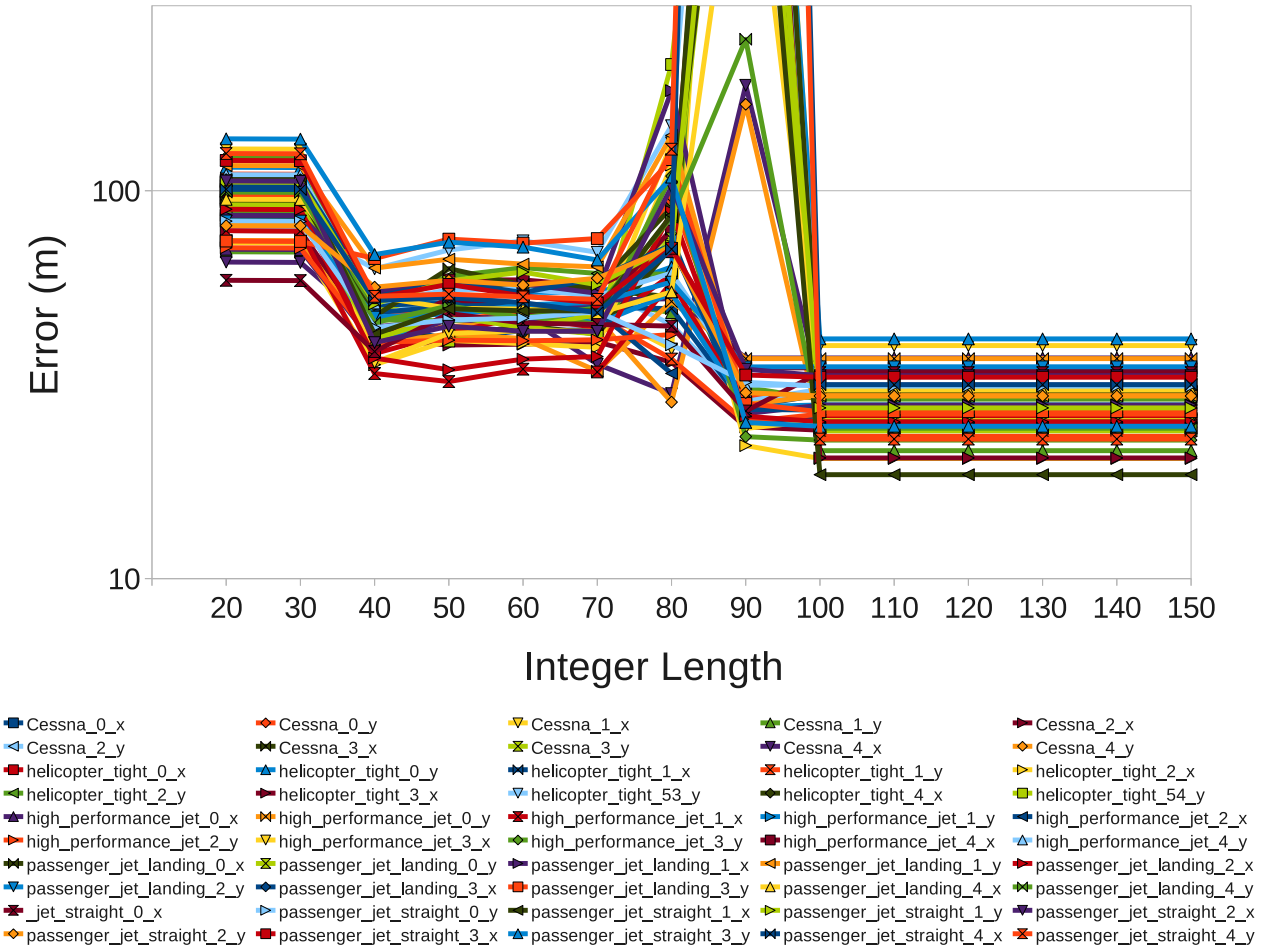


Figure B.1: Accuracy of position estimates: X and Y coordinates versus integer bit width.

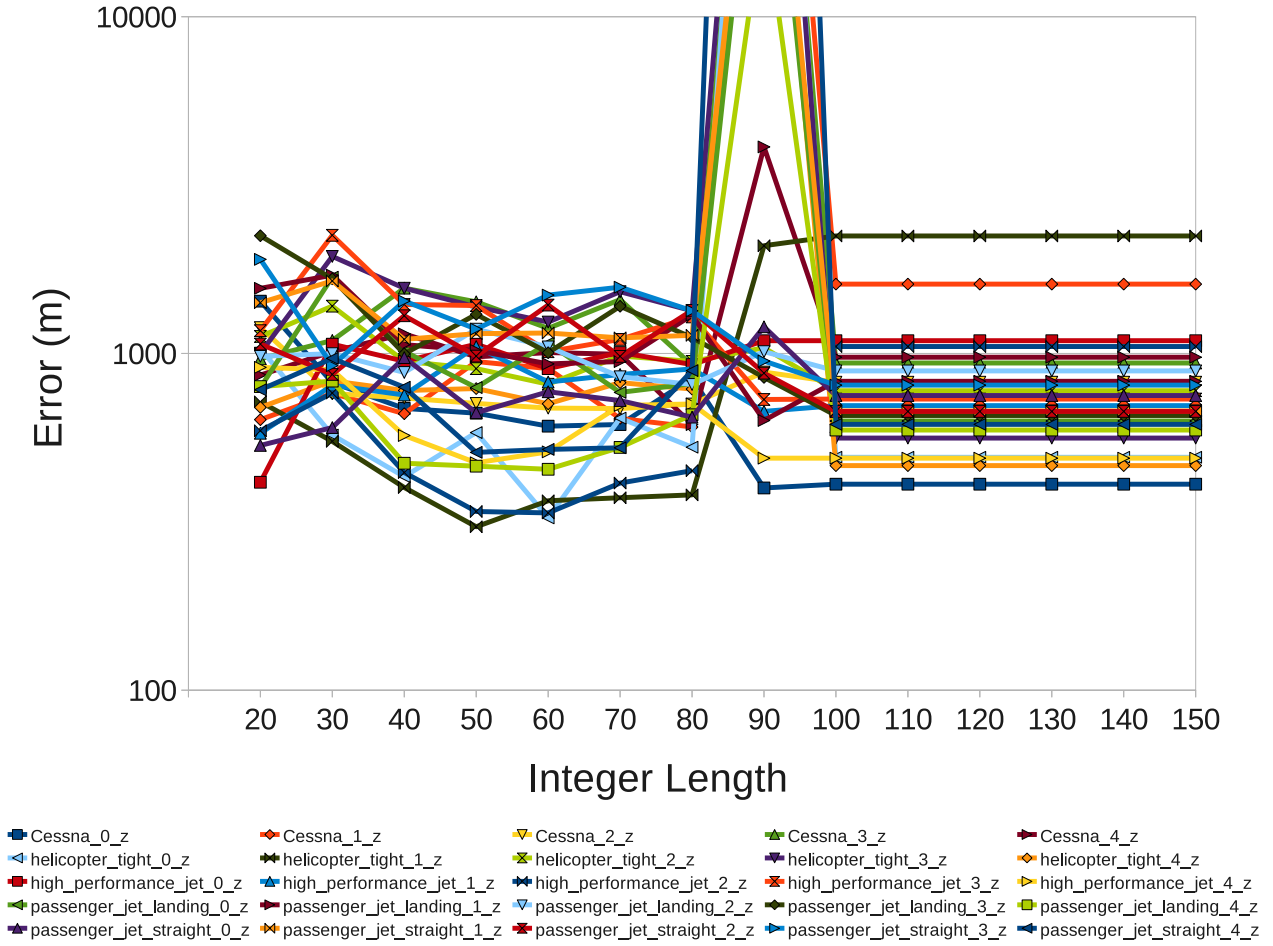


Figure B.2: Accuracy of position estimates: Z coordinate versus integer bit width. Note that some of the simulations ("helicopter_1" and "Cessna_1") performed worse after the 'knee' of the graph. As all the other simulations perform better, we went with the majority for the best statistical performance.

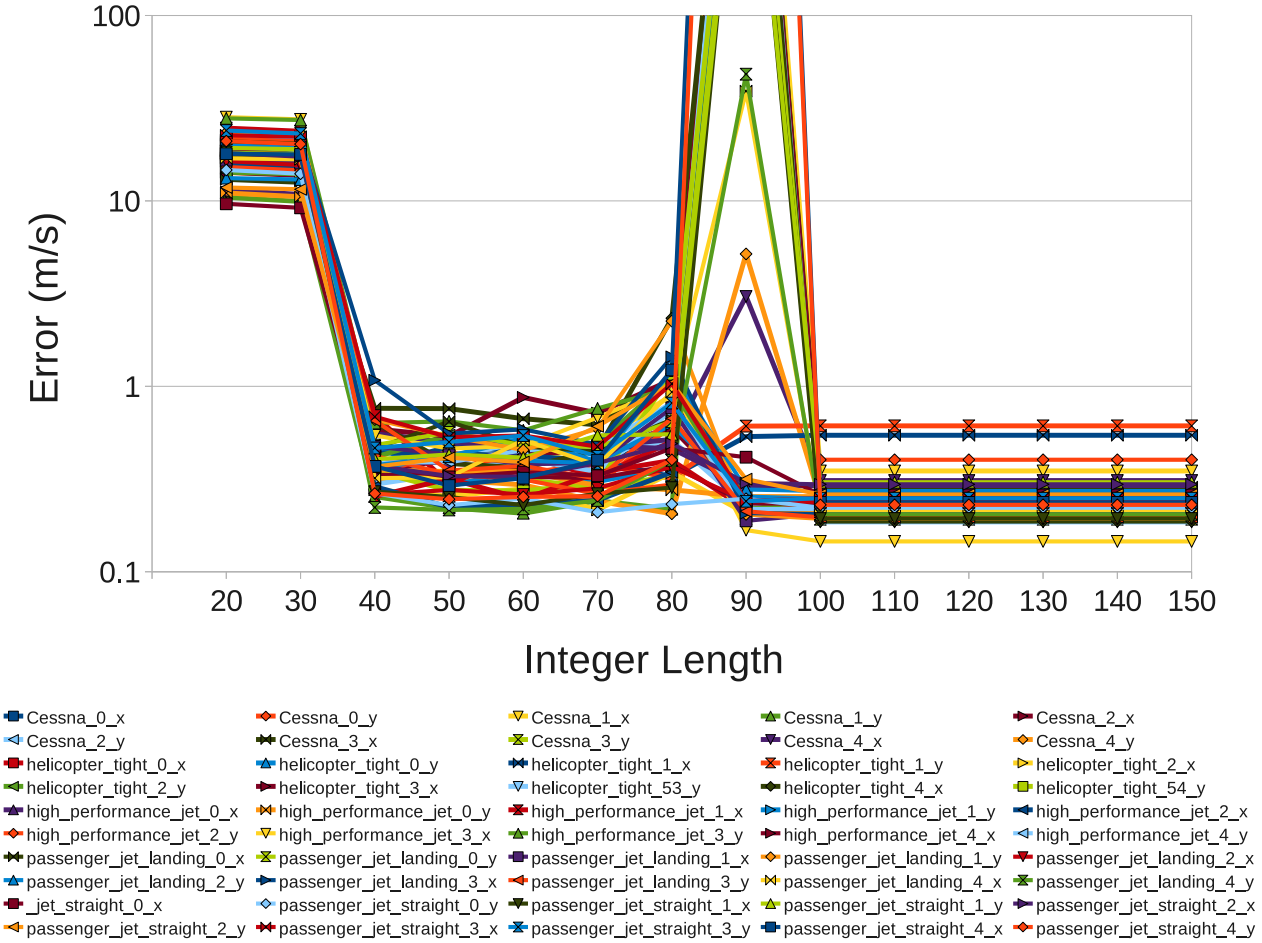


Figure B.3: Accuracy of velocity estimates: X and Y coordinates versus integer bit width.

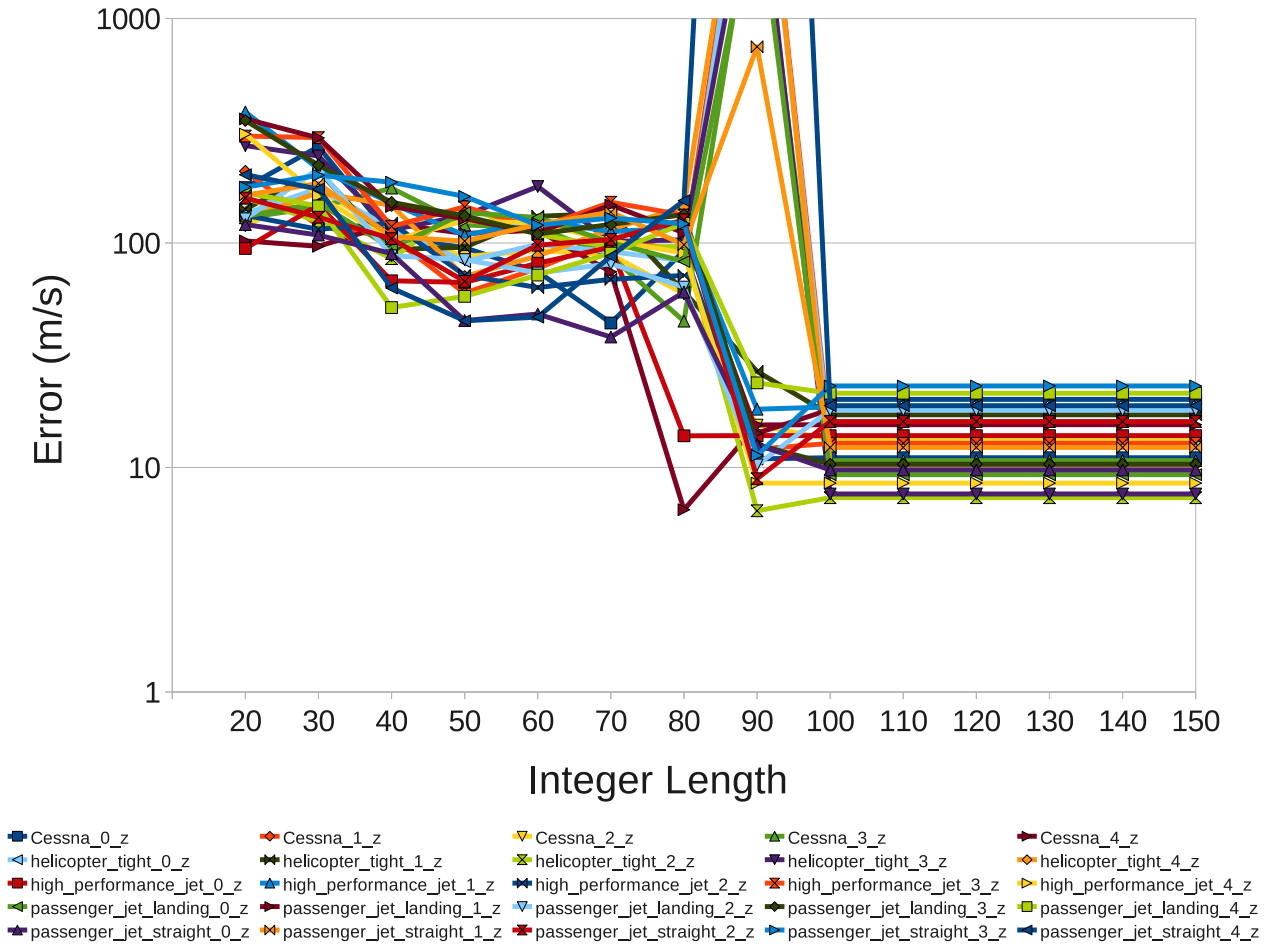


Figure B.4: Accuracy of velocity estimates: Z coordinate versus integer bit width.

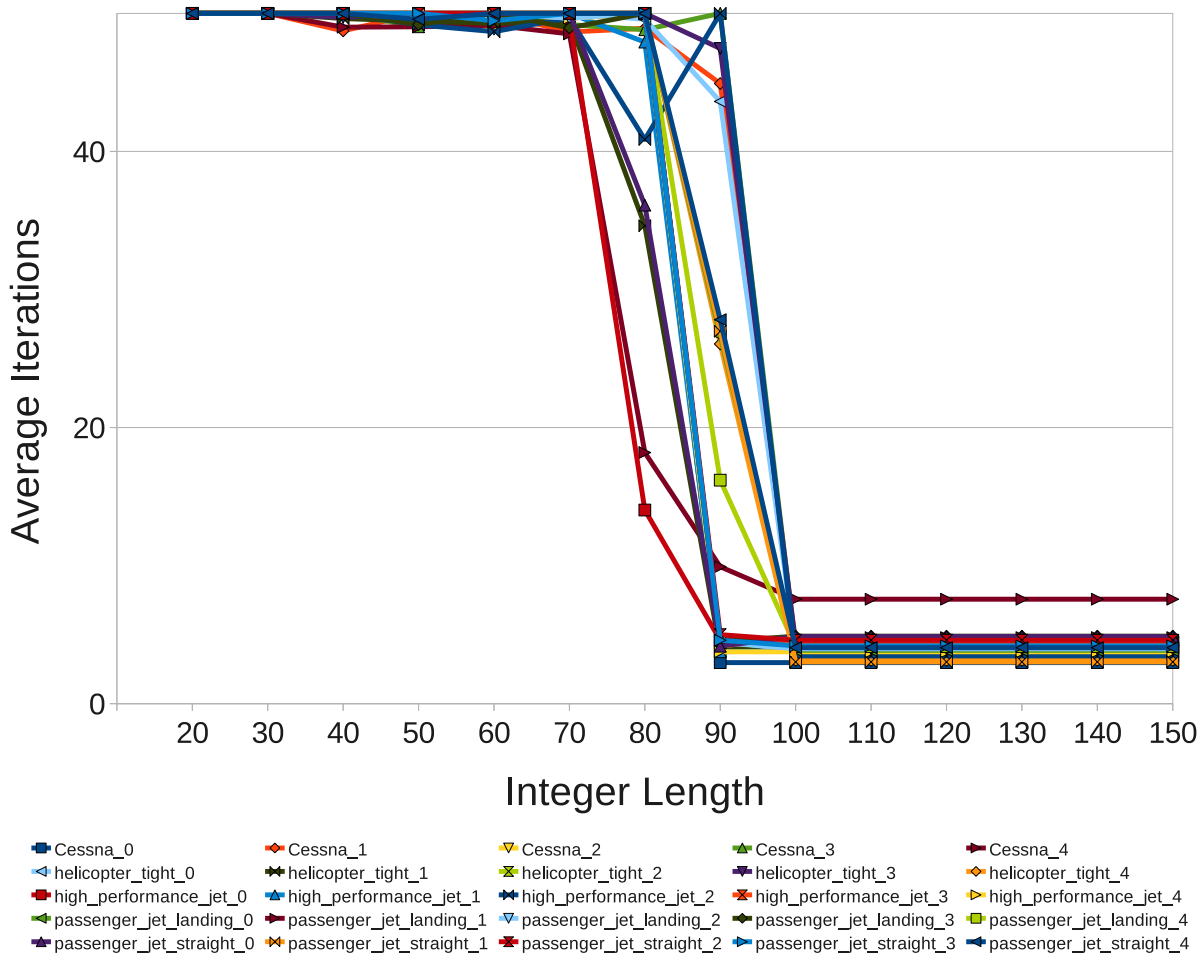


Figure B.5: Average iterations to convergence versus integer width. The imposed maximum was 50, and the software average was 4.5.

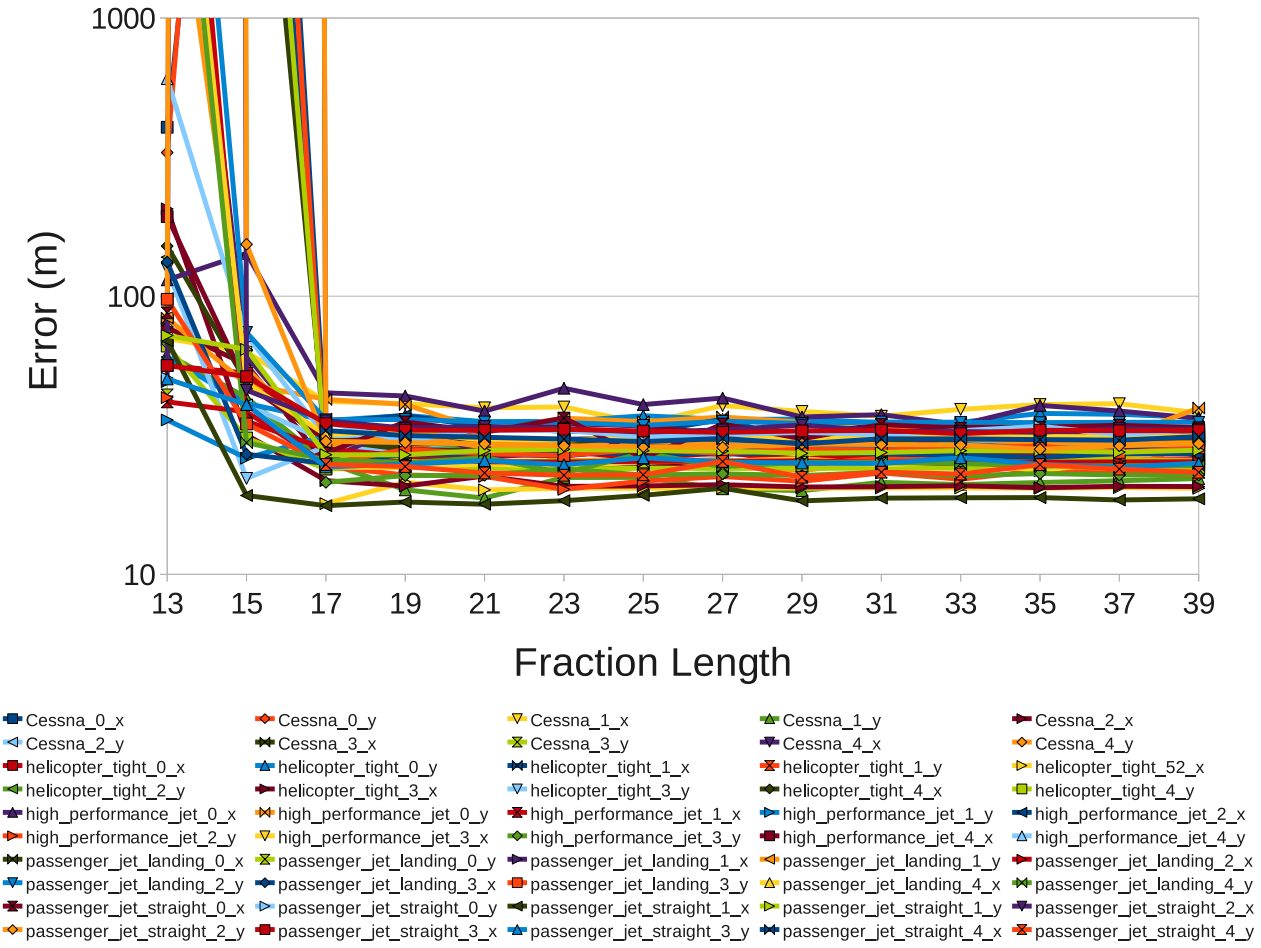


Figure B.6: Accuracy of position estimates: X and Y coordinates versus fraction bit width. The blank results for small fraction sizes represent NaNs, which occurred when the algorithm did not converge.

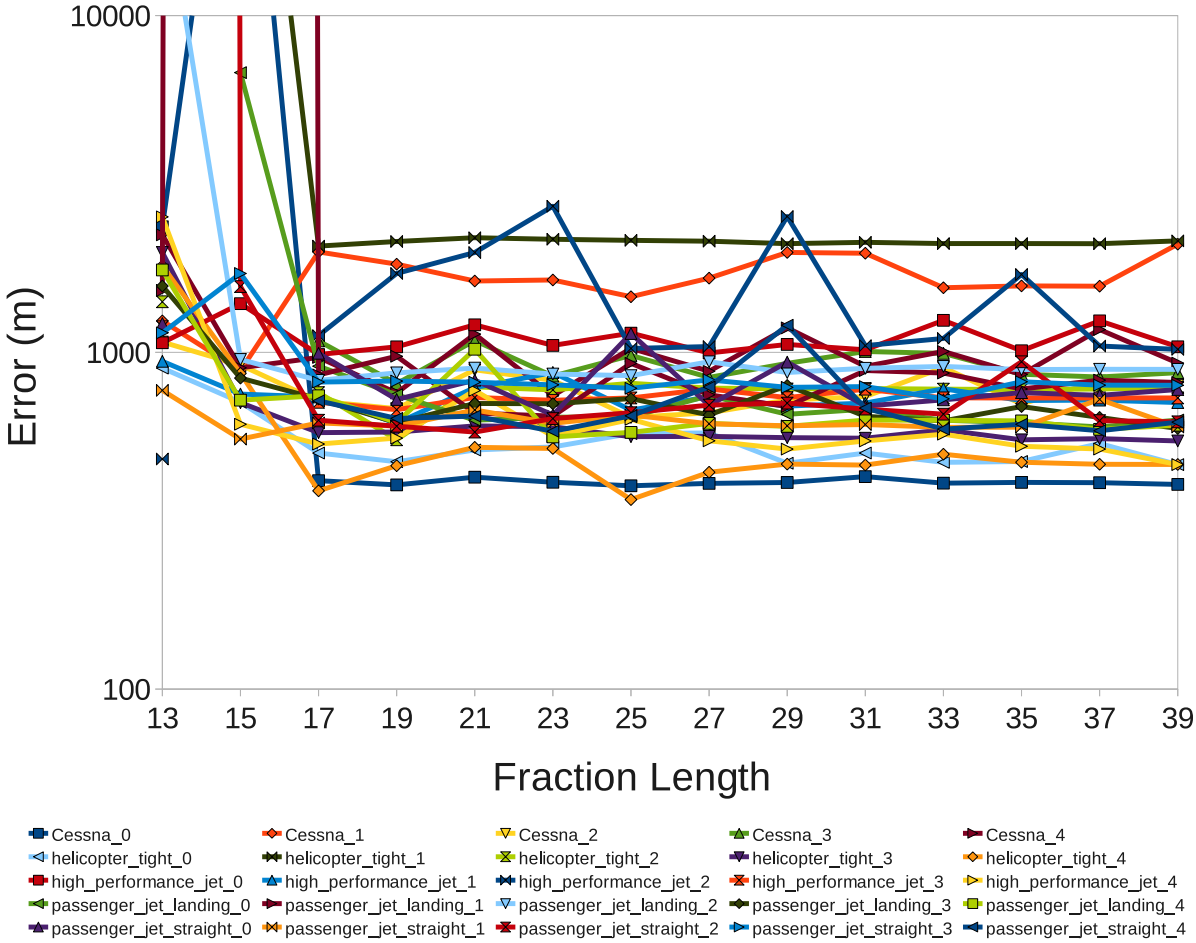


Figure B.7: Accuracy of position estimates: Z coordinate versus fraction bit width.

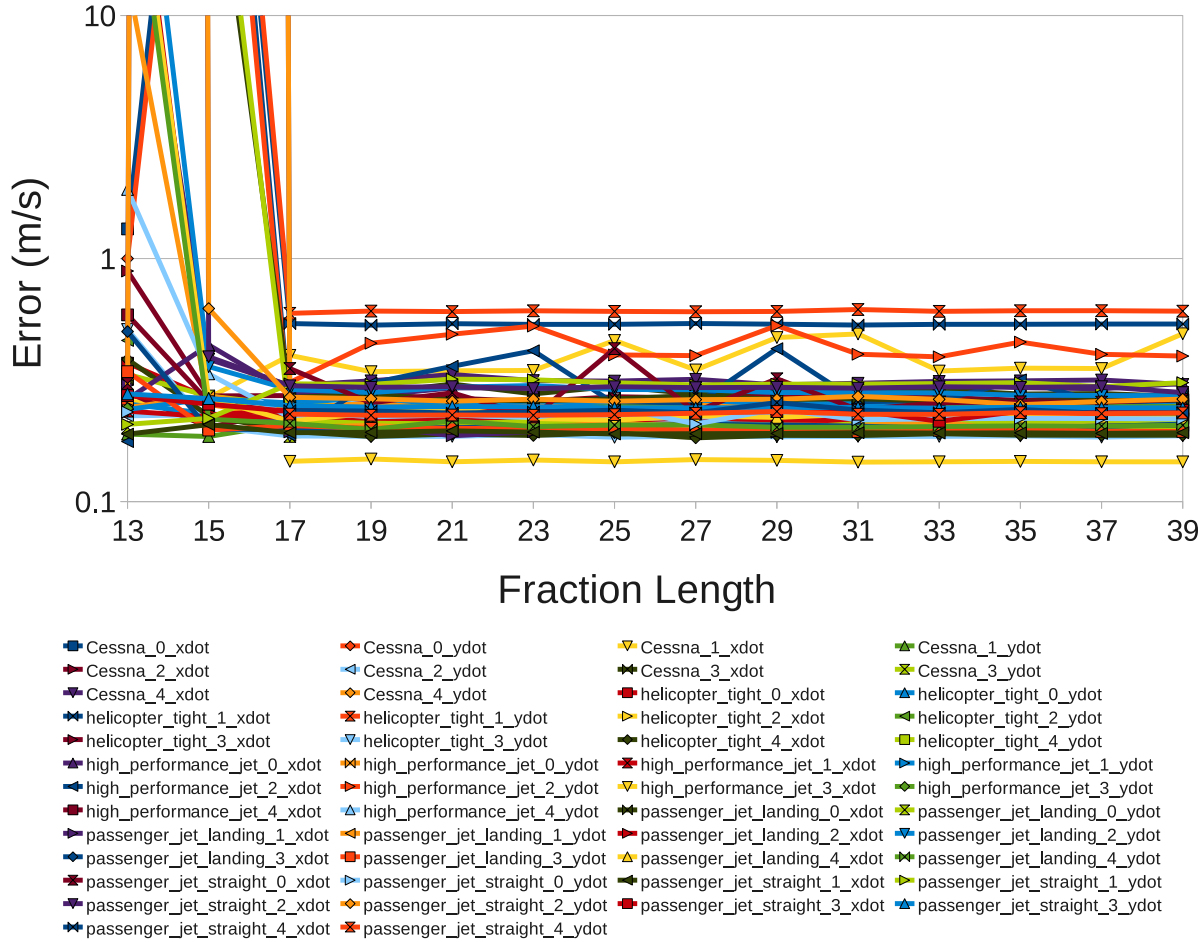


Figure B.8: Accuracy of velocity estimates: X and Y coordinates versus fraction bit width.

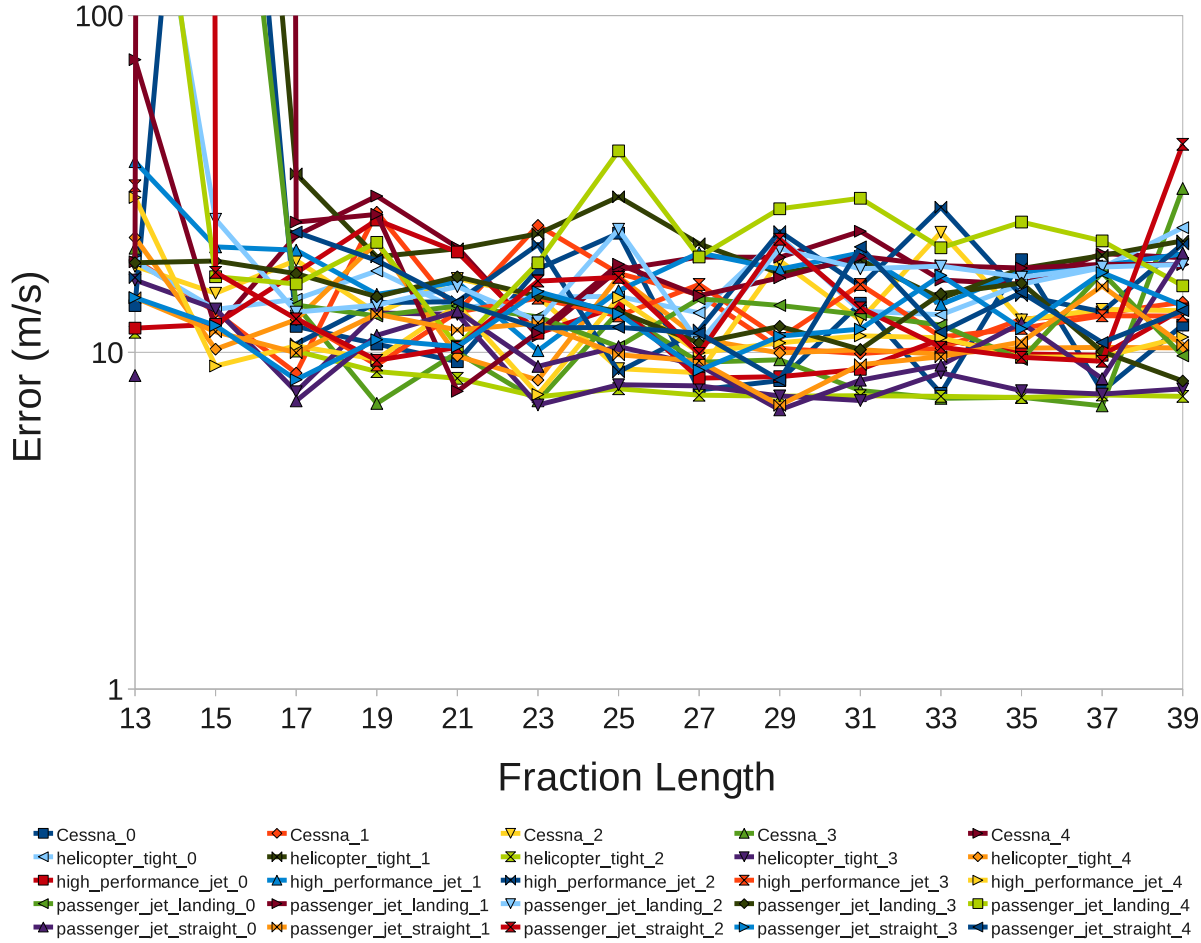


Figure B.9: Accuracy of velocity estimates: Z coordinate versus fraction bit width.

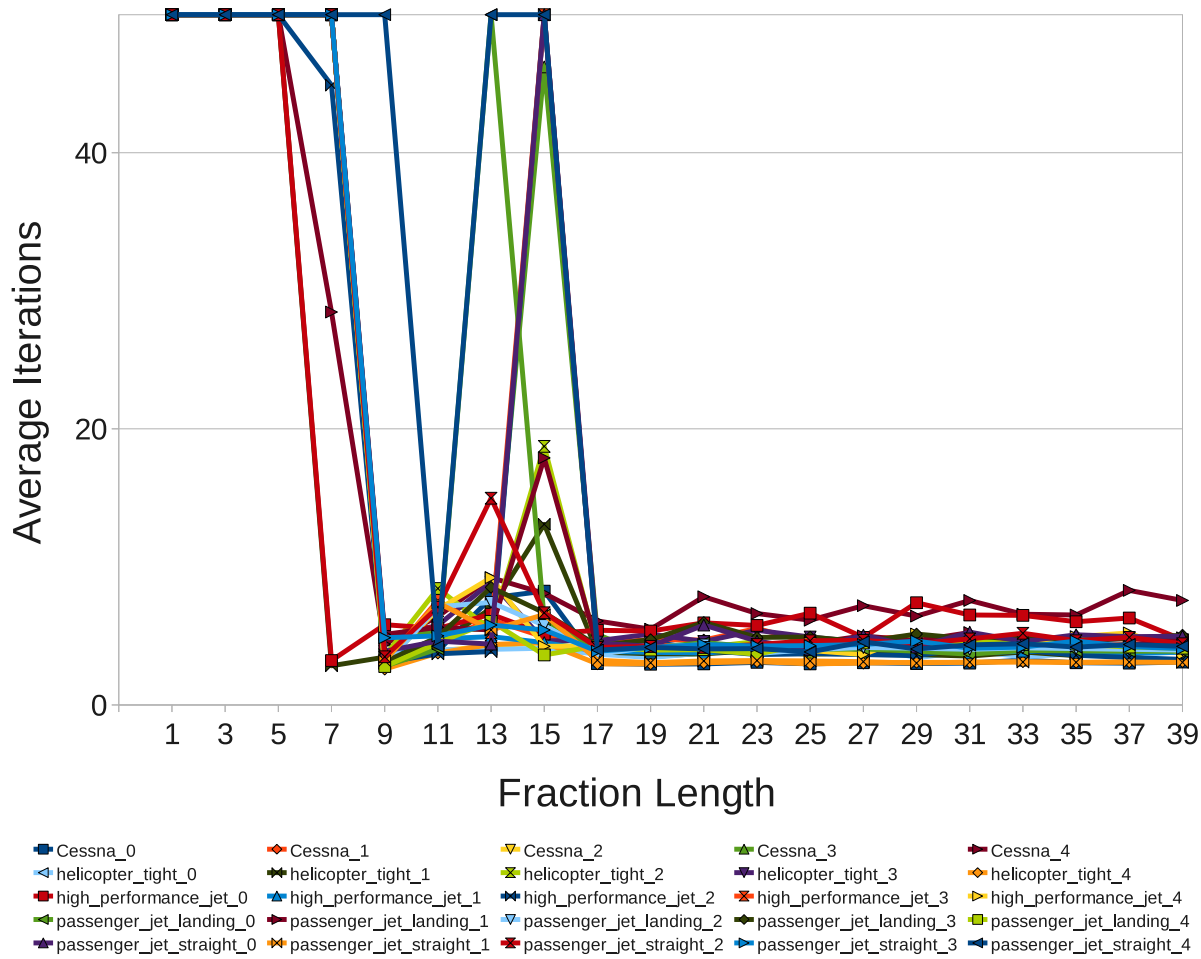


Figure B.10: Average iterations to convergence versus fraction length.

Appendix C

Python Scripts

This appendix provides listings of the various python scripts used to aid in profiling the algorithm and obtaining data from the software and hardware implementations of it.

C.1 Reduced-Precision Simulation

This script calls the C code simulation for a range of precisions and stores the accuracy results in a spreadsheet. Results are a comparison of the outputs of the simulation with the true states of the target, which are stored along with the synthetic radar data by the IDL data generator.

```
1  #!/usr/bin/env python
2
3  import sys
4  import os
5  import re
6  from time import gmtime, strftime
7  from xlwt import *
8
9  #exp = sys.argv[1]
10 #mant = sys.argv[2]
11
12 #TWEAKABLES
13 trueCompare = 0 # set this to 0 to compare with joe's code results,
14         #set to 1 to compare with TRUEstate results.
15 #/TWEAKABLES
16 if trueCompare:
17     print "Comparing with the true states.\n"
18 else:
19     print "Comparing with joe's code.\n"
20
21 #set up directories and files:
```

```
22 outputfile = open("./differences.txt", "w")
23 thetime = strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
24 outputfile.write("output for script ran on " + thetime + "\n\n")
25 outputfile.write("----------------------------------------------------\n")
26 outputfile.close()
27
28 wb = Workbook()
29 ws = wb.add_sheet("checking out c code")
30 ws.write(0,0,"mantissa length")
31 ws.write(0,1,"x error")
32 ws.write(0,2,"y error")
33 ws.write(0,3,"z error")
34 ws.write(0,4,"x_dot error")
35 ws.write(0,5,"y_dot error")
36 ws.write(0,6,"z_dot error")
37 ws.write(0,7,"iterations")
38 ws.write(2,10,"all averages")
39 ws.write(32,10,"all maxes")
40 ws.write(62,10,"all mins")
41 wb.save("c code errors.xls")
42
43 os.system("./diffcorr 1 23")
44 if(os.path.isdir("./normal_results")):
45     os.system("rm -r normal_results")
46 os.system("mkdir normal_results")
47 os.system("cp -r ./statevector ./normal_results/")
48 os.system("cp iterations.txt ./normal_results/")
49
50 #run prog for all different bitwidths
51 outputfile = open("./differences.txt", "a")
52
53 sidecounter = 3
54 row = 1
55 for i in range(0, 24): # over all mantissa lengths
56
57     outputfile.write("\nfor a mantissa width of " + str(i) + "\n")
58     outputfile.write("----------------------------------------------------\n\n")
59     ws.write(row, 0, str(i))
60
61     os.system("./diffcorr 1 " + str(i))
62     # paste the iteration stuff in
63     iterationsFile = open("./iterations.txt", "r")
64     iterationsLines = iterationsFile.readlines();
65     iterationsFile.close();
66     itrow = 0;
67     itrowave = 0;
```

```
68
69      max = [0, 0, 0, 0, 0, 0]
70      min = [999, 999, 999, 999, 999, 999]
71      sum = [0, 0, 0, 0, 0, 0]
72
73      for j in range(0, 37): # over all legs
74
75          #open the proper files and get data:
76          if j < 10:
77              bufchar = "0"
78          else:
79              bufchar = ""
80
81          filename = "posbar" + bufchar + str(j) + ".txt"
82          dotfilename = "posbardot" + bufchar + str(j) + ".txt"
83
84          #This part decides whether to look at joe's results
85          #or at the true results.
86          if trueCompare:
87              truestate = "TRUEstate/"
88              truefilename = "postrue" + bufchar + str(j) + ".txt"
89              truedotfilename = "posdottrue" + bufchar + str(j) + ".txt"
90          else:
91              truestate = ""
92              truefilename = filename
93              truedotfilename = dotfilename
94
95          #print "comparing : " + "./normal_results/statevector/"\
96                      + truestate + "0/" + truefilename + "\n"
97          file = open("./normal_results/statevector/" + truestate\
98                      + "0/" + truefilename, "r")
99          lines = file.readlines()
100         file.close()
101
102         file = open("./normal_results/statevector/"\
103                     + truestate + "0/" + truedotfilename, "r")
104         dotlines = file.readlines()
105         file.close()
106
107         #print "and : " + "./statevector/0/" + filename + "\n"
108         file = open("./statevector/0/" + filename, "r")
109         wronglines = file.readlines()
110         file.close()
111
112         file = open("./statevector/0/" + dotfilename, "r")
113         dotwronglines = file.readlines()
```

```
114         file.close()
115
116         #get data using REGULAR EXPRESSIONS...
117         searcher = re.compile(r'^\s+(.?\d+.\d+)\s+(.?\d+.\d+)\s+(.?\d+.\d+)')
118                 #start, then one character,
119                 #then optional character (minus sign),
120                 #then string of numbers,
121                 #then anything (decimal point) then a string of numbers.
122                 #three groups of these.
123
124         oldresult = searcher.search(lines[79])
125         dotoldresult = searcher.search(dotlines[79])
126
127         result = searcher.search(wronglines[79])
128         dotresult = searcher.search(dotwronglines[79])
129
130         #do the subtraction and store results
131         firstnum =      str(abs(float(oldresult.group(1))
132                                 - float(result.group(1))))
133         secondnum =     str(abs(float(oldresult.group(2))
134                                 - float(result.group(2))))
135         thirdnum =      str(abs(float(oldresult.group(3))
136                                 - float(result.group(3))))
137
138         fourthnum =     str(abs(float(dotoldresult.group(1))
139                                 - float(dotresult.group(1))))
140         fifthnum =      str(abs(float(dotoldresult.group(2))
141                                 - float(dotresult.group(2))))
142         sixthnum =      str(abs(float(dotoldresult.group(3))
143                                 - float(dotresult.group(3))))
144
145         answers = [firstnum, secondnum, thirdnum,
146                    fourthnum, fifthnum, sixthnum]
147
148         outputfile.write(firstnum + "\t\t" + secondnum + "\t\t" + thirdnum\
149                         + "\t\t" + fourthnum + "\t\t" + fifthnum\
150                         + "\t\t" + sixthnum + "\n")
151
152         for k in range(0,6):
153             value = float(answers[k])
154             ws.write(row, k+1, value)
155             if(value > max[k]):
156                 max[k] = value
157             if(value < min[k]):
158                 min[k] = value
159             sum[k] = sum[k] + value
```

```
160
161             ws.write(row, 7, iterationsLines[itrow][:-1])
162             itrowave = itrowave + int(iterationsLines[itrow][:-1])
163             itrow = itrow + 1
164             row = row + 1
165
166         ws.write(row, 0, "Max")
167         for k in range(1,7):
168             ws.write(row, k, max[k-1])
169             ws.write(sidecounter + 31, k+8, max[k-1])
170         ws.write(row+1, 0, "Min")
171         for k in range(1,7):
172             ws.write(row+1, k, min[k-1])
173             ws.write(sidecounter + 61, k+8, min[k-1])
174         ws.write(row+2, 0, "Ave") 165          for k in range(1,7):
175             ws.write(row+2, k, sum[k-1]/36)
176             ws.write(sidecounter, k+8, sum[k-1]/36)
177         ws.write(row+2, 7, itrowave/36)
178
179         sidecounter = sidecounter + 1
180         row = row + 4
181         wb.save("c code errors.xls")
182
183 wb.save("c code errors.xls")
184 outputfile.close()
185
186 print "Done.\n"
```

C.2 FPGA Resource Usage Trials

To investigate the scaling of resource requirements with word sizes, a script was written to parametrise and build the design for a range of word widths. It did this by editing the text files which store the IP core parameter values, resynthesising them in turn, and then rebuilding the design as a whole.

```
1  #!/usr/bin/env python
2
3  #builds my ISE project for the given exponent and mantissa widths.
4
5  import sys
6  import os
7  from time import time
8
9  start = time()
10
11 exp = sys.argv[1]
12 mant = sys.argv[2]
13
14 #for normal partials
15 arithComps = ['float_mult', 'float_sub', 'float_add', 'float_div',
16                                     'float_mult_dsp', 'float_sqrt']
17 #'shift' length based on first latency,
18 #'shift_long' length based on second. sqrt latency based on last.
19 shiftComps = ['shift', 'shift_long', 'shift_dcalc',
20                     'Bshift', 'shift_addminusmult']
21
22 coregen = '/opt/Xilinx/11.1/ISE/bin/lin/coregen '
23 path_to_project = '/home/jp/ISE_projects/filters/General_FP_doppler_partials/'
24 ipcoreDir = path_to_project + 'ipcore_dir/'
25 GNConsts = path_to_project + 'GNConstants.vhd'
26
27 def getLatency(xcoFile):
28     try:
29         file = open(xcoFile, "r")
30         lines = file.readlines()
31         file.close()
32     except IOError:
33         print "IOError on file " + xcoFile + ". (scrape latencies section)\n"
34         sys.exit()
35
36     for line in lines:
37         if(line[0:14] ==         'CSET c_latency'):
38             Lat =   line[15:-1]
39     return Lat
40
```

```
41
42
43 #generate arithmetic cores-----------------------------------
44 for comp in arithComps:
45     xcoFile = '/home/jp/ISE_projects/filters\
46 /General_FP_doppler_partials/ipcore_dir/' + comp + '.xco'
47     #edit its .xco file
48     try:
49         file = open(xcoFile, "r")
50         lines = file.readlines()
51         file.close()
52     except IOError:
53         print "*******************************IOError on file " + xcoFile\
54                                     + ".(generate arith cores section)\n"
55         sys.exit()
56
57     j = 0
58     theplaces = [999,999,999,999]
59     for line in lines:
60         if(line[0:23] ==         'CSET c_a_exponent_width'):
61             theplaces[0] = j
62         if(line[0:23] ==         'CSET c_a_fraction_width'):
63             theplaces[1] = j
64         if(line[0:28] ==         'CSET c_result_exponent_width'):
65             theplaces[2] = j
66         if(line[0:28] ==         'CSET c_result_fraction_width'):
67             theplaces[3] = j
68         j = j + 1
69
70     if(theplaces[0] > 900 or theplaces[1] > 900
71     or theplaces[2] > 900 or theplaces[3] > 900):
72         print '*****PARSE ERROR ON ARITHMETIC XCO FILE ' + comp
73
74     lines[theplaces[0]] = 'CSET c_a_exponent_width='\
75                           + exp + '\n'
76     lines[theplaces[1]] = 'CSET c_a_fraction_width='\
77                           + str(int(mant) + 1) + '\n'
78     lines[theplaces[2]] = 'CSET c_result_exponent_width='\
79                           + exp + '\n'
80     lines[theplaces[3]] = 'CSET c_result_fraction_width='\
81                           + str(int(mant) + 1) + '\n'
82
83     try:
84         file = open(xcoFile, "w")
85         file.writelines(lines)
86         file.close()
```

```
87      except IOError:
88          print "**************IOError on file " + xcoFile +
89                  ". (generate arith cores section)\n"
90          sys.exit()
91
92      #generate the core
93      command =  coregen + '-b' + ' ' + xcoFile + ' -p ' +  ipcoreDir +' -r'
94      print command
95      os.system(command)
96
97  #get the latency of the mult and sub components,
98  #shifters must shift this depth.
99  shortLat = ''
100 longLat = ''
101 DLat = ''
102 BLat = ''
103 minusLat = ''
104 DefValue = '' #shifter needs default data to be same length as word width
105
106 #scrape values
107 xcoFile = ipcoreDir + arithComps[0] + '.xco'
108 shortLat = getLatency(xcoFile)
109
110 xcoFile = ipcoreDir + arithComps[1] + '.xco'
111 longLat = getLatency(xcoFile)
112
113 minusLat = str(int(longLat) - int(shortLat))
114 #DLat =
115
116 #create strings for default data
117 print int(exp) + int(mant) + 1
118 for it in range(0,(int(exp) + int(mant) + 1)):
119     DefValue = DefValue + '0'
120 #generate shifter cores###################################
121 latencies = [shortLat, longLat]
122 i = 0
123 for comp in shiftComps:
124     xcoFile = ipcoreDir + comp + '.xco'
125     #edit its .xco file
126     try:
127         file = open(xcoFile, "r")
128         lines = file.readlines()
129         file.close()
130     except IOError:
131         print "IOError on file " + xcoFile\
132                 + ".(generate shifter cores section)\n"
```

```
133         sys.exit()
134
135     j = 0
136     theplaces = [999,999,999,999,999]
137     for line in lines:
138         if(line[0:17] ==        'CSET asyncinitval'):
139             theplaces[0] = j
140         if(line[0:16] ==        'CSET defaultdata'):
141             theplaces[1] = j
142         if(line[0:10] ==        'CSET depth'):
143             theplaces[2] = j
144         if(line[0:16] ==        'CSET syncinitval'):
145             theplaces[3] = j
146         if(line[0:10] ==        'CSET width'):
147             theplaces[4] = j
148         j = j + 1
149
150     if(theplaces[0] > 900 or theplaces[1] > 900 or
151         theplaces[2] > 900 or theplaces[3] > 900 or theplaces[4] > 900):
152         print '*********PARSE ERROR ON SHIFTER XCO FILE ' + comp
153
154     lines[theplaces[2]] = 'CSET depth=' + latencies[i] + '\n'
155     lines[theplaces[4]] = 'CSET width=' + str(int(mant) + int(exp) + 1) + '\n'
156     lines[theplaces[0]] = 'CSET asyncinitval=' + DefValue + '\n'
157     lines[theplaces[1]] = 'CSET defaultdata=' + DefValue + '\n'
158     lines[theplaces[3]] = 'CSET syncinitval=' + DefValue + '\n'
159
160     try:
161         file = open(xcoFile, "w")
162         file.writelines(lines)
163         file.close()
164     except IOError:
165         print "IOError on file " + xcoFile\
166                 + ".(generate shifter cores section)\n"
167         sys.exit()
168
169     #generate the core
170     command =  coregen + '-b' + ' ' + xcoFile + ' -p ' +  ipcoreDir +'-r'
171     print command
172     os.system(command)
173     i = i+1
174
175 #edit the project vhdl to agree with these sizes:-------------------
176 try:
177     file = open(GNConsts, "r")
178     lines = file.readlines()
```

```
179 except IOError:
180     print "IOError on file " + GNConsts + ".\n"
181     sys.exit()
182 
183 j = 0
184 theplaces = [999,999]
185 for line in lines:
186     if(line[0:15] == '   constant EXP'):
187         theplaces[0] = j
188     if(line[0:16] == '   constant MANT'):
189         theplaces[1] = j
190     j = j+1
191 
192 if(theplaces[0] > 900 or theplaces[1] > 900):
193     print '*********PYTHON SCRIPT ERROR - PARSE ERROR ON GNCONSTANTS FILE'
194 
195 lines[theplaces[0]] =\
196  '   constant EXP        : integer := ' + exp + '; --exponent width\n'
197 lines[theplaces[1]] =\
198   '   constant MANT       : integer := ' + mant + '; --mantissa width\n'
199 
200 file.close()
201 try:
202     file = open(GNConsts, "w")
203     file.writelines(lines)
204     file.close()
205 except IOError:
206     print "IOError on file " + GNConsts + ".\n"
207     sys.exit()
208 
209 #run ise -synthesis
210 os.chdir(path_to_project)
211 path_to_dotise_file = '"' + path_to_project\
212                           + 'General_FP_doppler_partials.ise"'
213 path_to_dotxst_file = '"' + path_to_project\
214                           + 'Get_Doppler_partial.xst"'
215 path_to_dotsyr_file = '"' + path_to_project\
216                           + 'Get_Doppler_partial.syr"'
217 command = 'xst -ise ' + path_to_dotise_file + ' -intstyle ise -ifn '\
218                           + path_to_dotxst_file\
219                           + ' -ofn ' + path_to_dotsyr_file
220 os.system(command)
221 
222 #run ise -translate, map, place & route (implement design
223 os.chdir(path_to_project)
224 
```

```
225 os.system("ngdbuild -ise " + path_to_dotise_file
226             + " -intstyle ise -dd _ngo -sd ipcore_dir -nt timestamp\
227 -i -p xc4vlx100-ff1148-12 Get_Doppler_partial.ngc  Get_Doppler_partial.ngd")
228
229 os.system("map -ise " + path_to_dotise_file
230             + " -intstyle ise -p xc4vlx100-ff1148-12 -global_opt off\
231 -cm area -ir off -pr off -c 100 -o Get_Doppler_parti \
232 al_map.ncd Get_Doppler_partial.ngd Get_Doppler_partial.pcf")
233
234 os.system("par -ise " + path_to_dotise_file
235             + " -w -intstyle ise -ol std -t 1 Get_Doppler_partial_map.ncd \
236 Get_Doppler_partial.ncd Get_Doppler_partial.pcf    ")
237
238 os.system("trce -ise " + path_to_dotise_file
239             + " -intstyle ise -v 3 -s 12 -fastpaths -xml \
240 Get_Doppler_partial.twx \
241 Get_Doppler_partial.ncd -o Get_Doppler_    partial.twr Get_Doppler_partial.pcf")
242
243 # scrape logs and append to summary
244 os.chdir(path_to_project)
245 end = time()
246 duration = str(end - start)
247 try:
248     file = open("summary.log", "a")
249         file.write("-------------------------------\n\n")
250     file.write("Running for exponent: " + exp + " and mantissa: " +
251             mant + ".  Process took "+ duration +" seconds. \n\n")
252     file.close()
253
254 except IOError:
255     print "***************problem opening summary file."
256     sys.exit()
257
258 os.system("./ScrapeLogs.py")
```

Another script was then called which scraped the log files for the information of interest.

```
#!/usr/bin/env python
import os
import glob
from xlwt import *

#####################################################
# define the deed...
def scrapeFiles(location):

    checker = 0
    clkFreq = ''

    # Open the synth file for read-only
    synthFile = glob.glob(location + '/*.syr')
    if synthFile:
        synthFile = synthFile[0]
        print "    found file " + synthFile
        file = open(synthFile, "r")
        lines = file.readlines()
        # Loop through the lines to find the right line
        i = -1
        for line in lines:
            i = i + 1
            if (line[0:19] == 'Clock Signal       '):
                    checker = checker + 1
                    print "        got clock load..."
                    clkLoad = lines[i + 2][62:-2]
            if (line[0:19] == '   Minimum period: '):
                    checker = checker + 1
                    print "        got clock frequency..."
                    clkFreq = line[28:-2]
        # Close the file
        file.close()
    else:
        print "    no synthesis file in this directory."

    # Open the implementation file for read-only
    impleFile = glob.glob(location + '/*.mrp')
    if impleFile:
        impleFile = impleFile[0]
        print "    found file " + impleFile

        file = open(impleFile, "r")
        lines = file.readlines()
```

```
45          # Loop through the lines to find the right line
46          i = -1
47          for line in lines:
48
49              i = i + 1
50              if (line[0:29] == '  Number of Slice Flip Flops:'):
51                  checker = checker + 1
52                  slcFlpFlps = line
53                  print "        got slice Flip Flops..."
54                  LUTs = lines[i+1]
55                  print "        got 4-input LUTS..."
56                  occSlice = lines[i + 3]
57                  print "        got occupied slices..."
58                  logSlice = lines[i + 4]
59                  print "        got logic slices..."
60                  unrSlice = lines[i + 5]
61                  print "        got unrelated slices..."
62                  totLUT = lines[i + 7]
63                  print "        got total LUTs..."
64                  logLUT = lines[i + 8]
65                  print "        got logic LUTs..."
66                  rtLUT = lines[i + 9]
67                  print "        got route LUTs..."
68                  shiftLUT = lines[i + 10]
69                  print "        got shift register LUTS..."
70              if (line[0:33] == 'Average Fanout of Non-Clock Nets:'):
71                  checker = checker + 1
72                  aveNet = line
73
74          # Close the file
75          file.close()
76      else:
77          print "    no implementation file in this directory."
78
79      # Open the second implementation file for read-only
80      impleFile2 = glob.glob(location + '/*.par')
81      if impleFile2:
82          impleFile2 = impleFile2[0]
83          print "    found file " + impleFile2
84
85          file = open(impleFile2, "r")
86          lines = file.readlines()
87          # Loop through the lines to find the right line
88          i = -1
89          for line in lines:
90
```

```
91              i = i + 1
92              if (line[0:27] == 'Device Utilization Summary:'):
93                  checker = checker + 1
94                  BUFGs = lines[i + 2]
95                  print "        got BUFGs...."
96                  DSP48 = lines[i + 3]
97                  print "        got DSPs..."
98                  EIOBs = lines[i + 4]
99                  print "        got external IOBs..."
100                 LIOBs = lines[i + 5]
101                 print "        got LOCed IOBs..."
102                 RAM16 = lines[i + 7]
103                 print "        got block RAMs..."
104                 slics = lines[i + 8]
105                 print "        got slices..."
106                 SLICM = lines[i + 9]
107                 print "        got SLICEMs..."
108             if (line[0:30] == '|         Clock Net    |    Reso'):
109                 checker = checker + 1
110                 fanout = lines[i + 2][46:51]
111                 print "        got fanout."
112             else:
113                 fanout = "XXX"
114         # Close the file
115         file.close()
116     else:
117         print "    no second implementation file in this directory."
118
119
120     # write data to file
121     file = open("summary.log", "a")
122     file.write("-------------------------------------------------\n")
123     if synthFile:
124
125         file.write("in file " + synthFile + ":\n\n")
126         file.write("clock load: \t\t" + clkLoad + "\n")
127         file.write(clkFreq + "\n\n")
128
129     if impleFile:
130         file.write("in file " + impleFile + ":\n\n")
131         # Save nice ones to list
132         dataList = [slcFlpFlps,LUTs,occSlice,
133                     totLUT,logLUT,rtLUT,shiftLUT,aveNet]
134
135         for datum in dataList:
136             file.write(datum[0:35] + "\t\t\t\t\t" + datum[36:-1] + "\n")
```

```
137             if (datum == occSlice):
138                 file.write(logSlice)
139                 file.write(unrSlice)
140     if impleFile2:
141         file.write("in file " + impleFile2 + ":\n\n")
142         # Save nice ones to list
143         dataList = [BUFGs,DSP48,EIOBs,LIOBs,RAM16,slics,SLICM]
144 
145         for datum in dataList:
146             file.write(datum[0:35] + "\t\t\t\t\t" + datum[36:-1] + "\n")
147             if (datum == occSlice):
148                 file.write(logSlice)
149                 file.write(unrSlice)
150 
151         file.write("clock fanout: \t\t\t\t\t\t\t\t\t\t" + fanout)
152         file.write("\n\n")
153     file.close()
154 
155     #check if it there were no obvious errors
156     if synthFile and impleFile:
157         if (checker == 6):
158             print "The script seems to have worked properly.\n"
159             return 0
160         else:
161             print "something went wrong.  Make sure everything is there.\n"
162             return 1
163     else:
164         return 0
165 
166 ############################################################
167 
168 numErrors = 0
169 
170 if (len(sys.argv) < 2):
171     location = os.getcwd()
172 else:
173     location = sys.argv[1]
174 
175 print "\nProcessing", location, "..."
176 numErrors = numErrors + scrapeFiles(location)
177 
178 if (numErrors == 0):
179     print "the Entire script ran with no errors.\n"
180 else:
181     print "there were errors - check out the results.\n"
```

C.3 Verification Scripts

For verification purposes, a script was created to compare simulation results with the results of performing the calculations in software. Inputs and outputs of the hardware simulation were text files containing binary representations of the data, and these had to be converted to decimal numbers for the comparison. Note that the function definitions used in this script are not included in this listing. For the full script, see the included DVD.

```
1  #!/usr/bin/env python
2  #this is automated testing using an ISE testbench with file
3  #I/O for floating_point project.
4  #you run the testbench in ISE then pass the input and output
5  #to this to check its accuracy.
6  #usage: ./testacc.py input_file output_file
7
8  import sys
9  import os
10 import math
11 from time import time
12
13 ################### Main Method ##################
14
15 start = time()
16 print ""
17
18 ############CHANGE THESE TO AGREE###############
19 expbits = 8
20 mantbits = 12
21
22 #read in all the data
23 inputfile = open(sys.argv[1], "r")
24 inlines = inputfile.readlines()
25 inputfile.close()
26
27 outputfile = open(sys.argv[2], "r")
28 outlines = outputfile.readlines()
29 outputfile.close()
30
31 #now we go through all the lines
32 #and subtract HW answer from python answer.
33 errorsum = Vector([0,0,0,0,0,0,0,0,0])
34 percenterrorsum = Vector([0,0,0,0,0,0,0,0,0])
35 allerrors = []
36 mins = [float('inf'), float('inf'), float('inf'),
37         float('inf'), float('inf'), float('inf'),
```

```
38          float('inf'), float('inf'), float('inf')]
39 maxes = [-1*float('inf'), -1*float('inf'), -1*float('inf'),
40              -1*float('inf'), -1*float('inf'), -1*float('inf'),
41              -1*float('inf'), -1*float('inf'), -1*float('inf')]
42
43 for i in range (0, len(outlines)):
44     inputs  = inlines[i].split()
45     A       = bin2dec(inputs[0], expbits, mantbits)
46     B       = bin2dec(inputs[1], expbits, mantbits)
47     x       = bin2dec(inputs[2], expbits, mantbits)
48     y       = bin2dec(inputs[3], expbits, mantbits)
49     z       = bin2dec(inputs[4], expbits, mantbits)
50     xdot    = bin2dec(inputs[5], expbits, mantbits)
51     ydot    = bin2dec(inputs[6], expbits, mantbits)
52     zdot    = bin2dec(inputs[7], expbits, mantbits)
53     xpos    = bin2dec(inputs[8], expbits, mantbits)
54     ypos    = bin2dec(inputs[9], expbits, mantbits)
55     zpos    = bin2dec(inputs[10], expbits, mantbits)
56
57     realResults = Vector(getRealAnswer(A, B,
58                                        x, y, z,
59                                        xdot, ydot, zdot,
60                                        xpos, ypos, zpos))
61
62     outputs         = outlines[i].split()
63     xdop            = bin2dec(outputs[0], expbits, mantbits)
64     ydop            = bin2dec(outputs[1], expbits, mantbits)
65     zdop            = bin2dec(outputs[2], expbits, mantbits)
66     xdotdop         = bin2dec(outputs[3], expbits, mantbits)
67     ydotdop         = bin2dec(outputs[4], expbits, mantbits)
68     zdotdop         = bin2dec(outputs[5], expbits, mantbits)
69     bear2           = bin2dec(outputs[6], expbits, mantbits)
70     beardcosdx      = bin2dec(outputs[7], expbits, mantbits)
71     beardsindy      = bin2dec(outputs[8], expbits, mantbits)
72     myResults       = Vector([xdop, ydop, zdop,
73                               xdotdop, ydotdop, zdotdop,
74                               bear2, beardcosdx, beardsindy])
75
76 #   print bin2hex(outputs[0] )
77 #   print bin2hex(outputs[1] )
78 #   print bin2hex(outputs[2] )
79 #   print bin2hex(outputs[3] )
80 #   print bin2hex(outputs[4] )
81 #   print bin2hex(outputs[5] )
82 #   print bin2hex(outputs[6] )
83 #   print bin2hex(outputs[7] )
```

```
84 #    print bin2hex(outputs[8] )
85
86      errors = myResults - realResults
87      errors = errors*errors  #note this does dot product,
88                              #not vecter multiply. defined in class above.
89      errornorm = errors.sqrt()
90      percenterror = errornorm/realResults.abs()
91      percenterror = percenterror*100
92
93 #    print A, B, x, y, z, xdot, ydot, zdot, xpos, ypos, zpos
94 #    print "Real results:"
95 #    print realResults
96 #    print "My results:"
97 #    print myResults
98 #    print "Error:"
99 #    print errornorm
100 #   print "% Error:"
101 #   print percenterror
102 #   print ""
103
104     allerrors.append(errornorm)
105     errorsum = (errorsum + errornorm)
106     percenterrorsum = (percenterrorsum + percenterror)
107     for j in range(0,len(maxes)):
108         if(errornorm[j] > maxes[j]):
109             maxes[j] = errornorm[j]
110         if(errornorm[j] < mins[j]):
111             mins[j] = errornorm[j]
112
113 print '\n\n'
114 aveError = errorsum/len(outlines)
115 avePercError = percenterrorsum/len(outlines)
116 print "average error:"
117 print aveError
118 print "average % error:"
119 print avePercError
120
121 stdDevErrorSum = Vector([0,0,0,0,0,0,0,0,0])
122 for oneerror in allerrors:
123     oneerror = oneerror - aveError
124     oneerror = oneerror*oneerror
125     stdDevErrorSum = stdDevErrorSum + oneerror
126
127 stdDevSquared = stdDevErrorSum/(max((len(outlines)-1),1))
128 stdDev = stdDevSquared.sqrt()
129 print "std devs:"
```

```
130 print stdDev
131
132 print "maximum error:"
133 print maxes
134 print "minimum error:"
135 print mins
136 print '\n\n'
```

Appendix D

Square-Root Shifting

In the fixed-point design, the IP core used for the square-root operation implements the CORDIC algorithm. It can be configured to calculate the square-root of a fixed-point input with exactly one integer bit. For the purposes of our design we thus need some reinterpretation of the data in order to use it for general fixed-point numbers.

As detailed in the core datasheet[60], the output scaling is determined as follows:

- The core calculates the root Y of input values X in the range $0 \leqslant X < 2$.

$$Y = \sqrt{X} \tag{D.1}$$

- The custom data format we wish to use will represent values in the range $0 \leqslant X_{alt} < 2^{N+1}$ where N is the index of the most significant bit.

$$Y_{alt} = \sqrt{X_{alt}} \tag{D.2}$$

- Interpreting X_{alt} using the standard core data format scales the input by $2^{-\text{N}}$.

$$Y = \sqrt{2^{-N} \cdot X_{alt}} \tag{D.3}$$

$$Y = 2^{(-N)/2} \cdot \sqrt{X_{alt}} \tag{D.4}$$

- When N is even the scaling factor is an integer power of two and is easily compensated for by a bit shift. However, when N is odd, an additional output scaling factor of $\sqrt{2}$ is introduced. To avoid having to multiply the output by this constant, we translate the $\sqrt{2}$ scaling to the input of the square-root function. If we set $2^{-N/2} = 2^{-M-(1/2)}$:

$$Y = 2^{(-M-1/2)} \cdot \sqrt{X_{alt}} \tag{D.5}$$

$$Y = 2^{-M} \cdot \sqrt{2^{-1} \cdot X_{alt}} \tag{D.6}$$

- Thus in order to use the core for a custom fixed-point number system we perform the following steps:
  - If N is odd, right-shift the input by one
  - pass the input to the core
  - If N is even, left-shift the output by N/2. Otherwise, left-shift the output by (N-1)/2

Appendix E

Detailed Error Comparisons

This section provides a more detailed look at the accuracy results of the three designs.

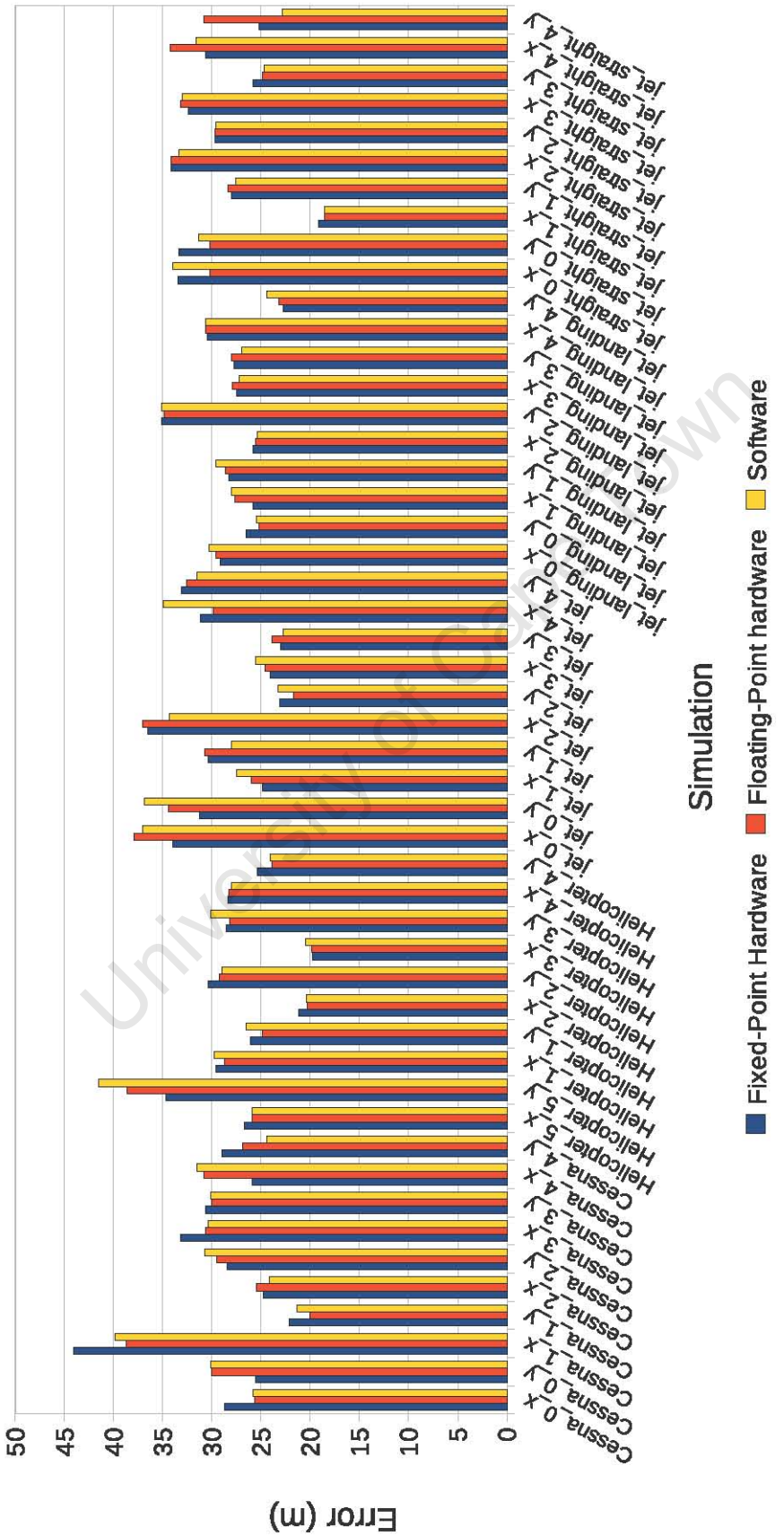


Figure E.1: Position estimate errors for the X and Y coordinates.

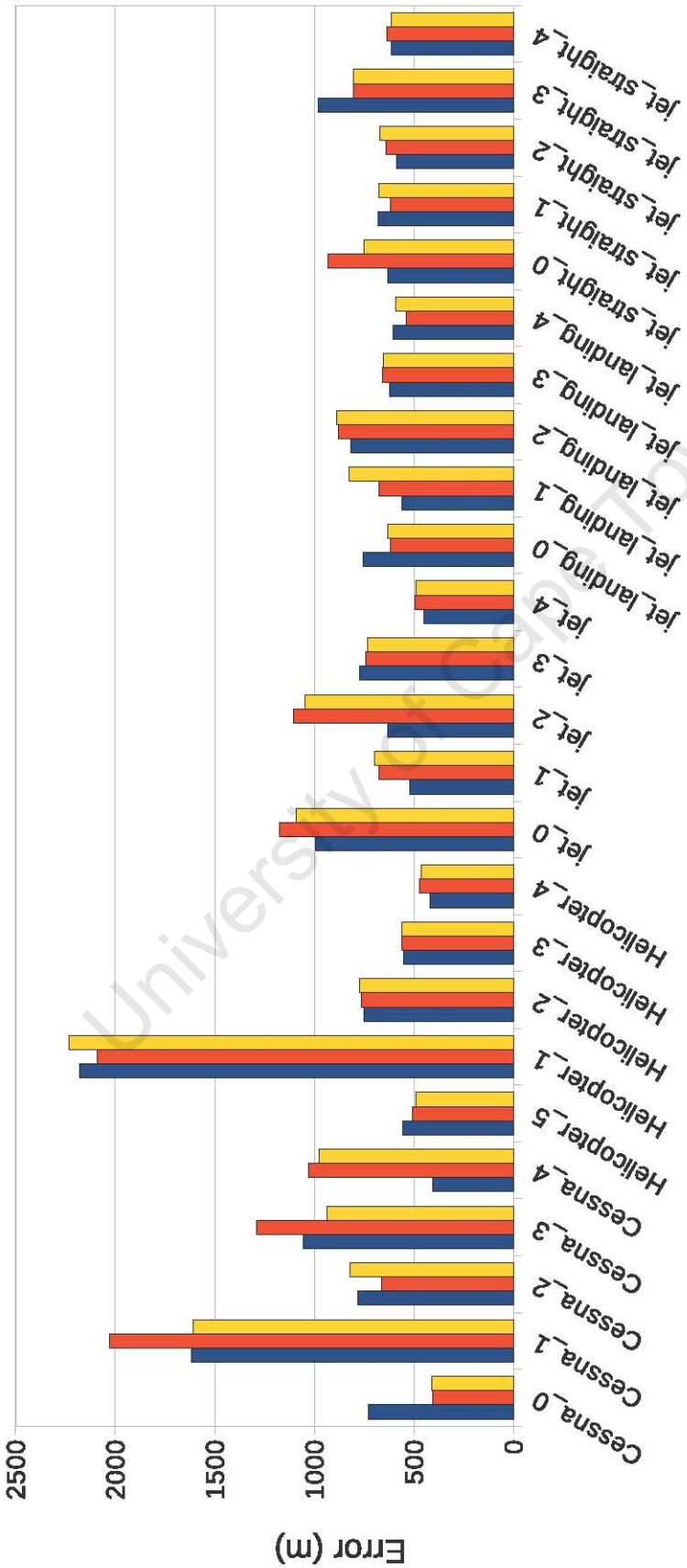


Figure E.2: Position estimate errors for altitude.

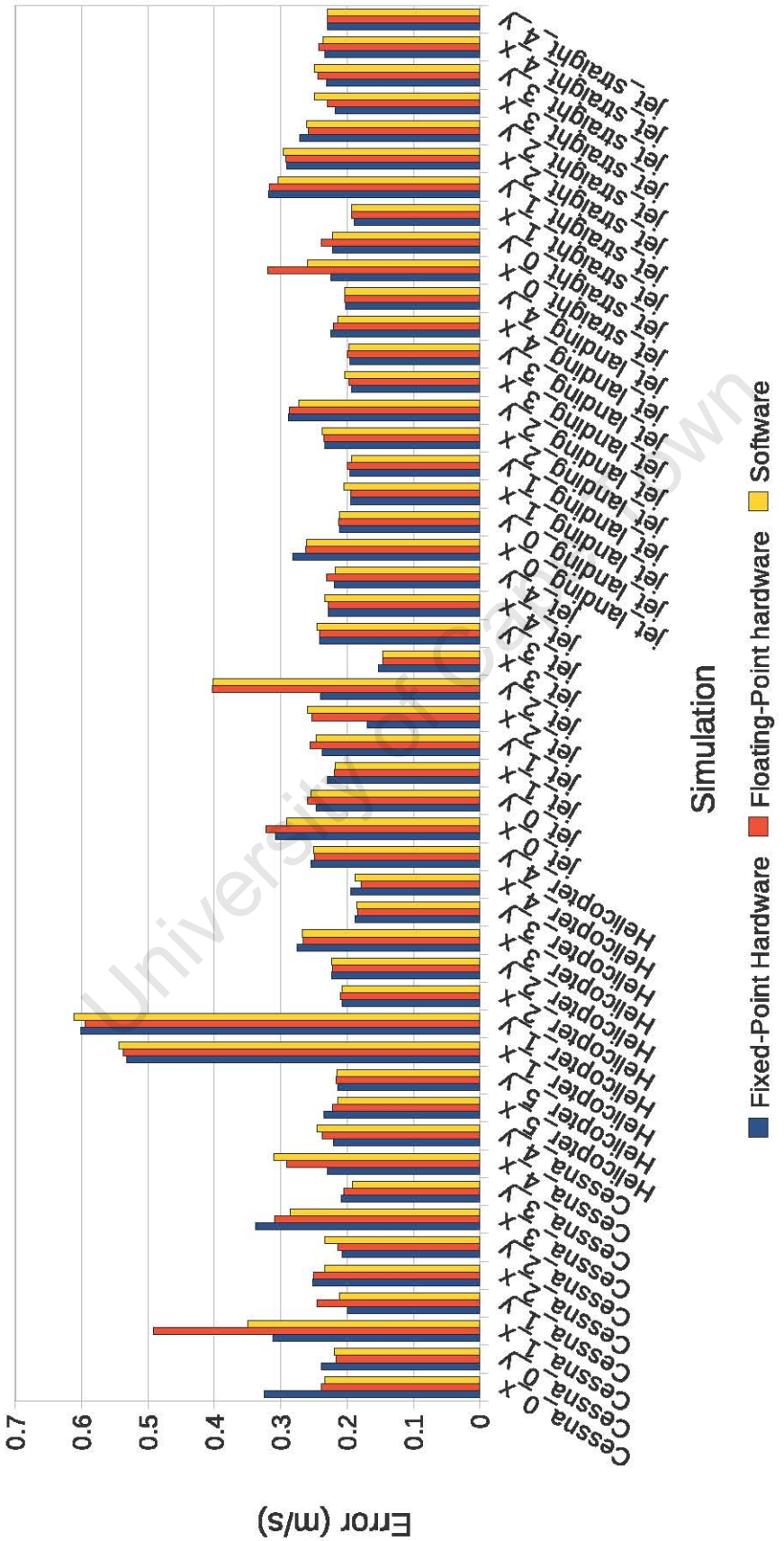


Figure E.3: Velocity estimate errors for X and Y coordinates

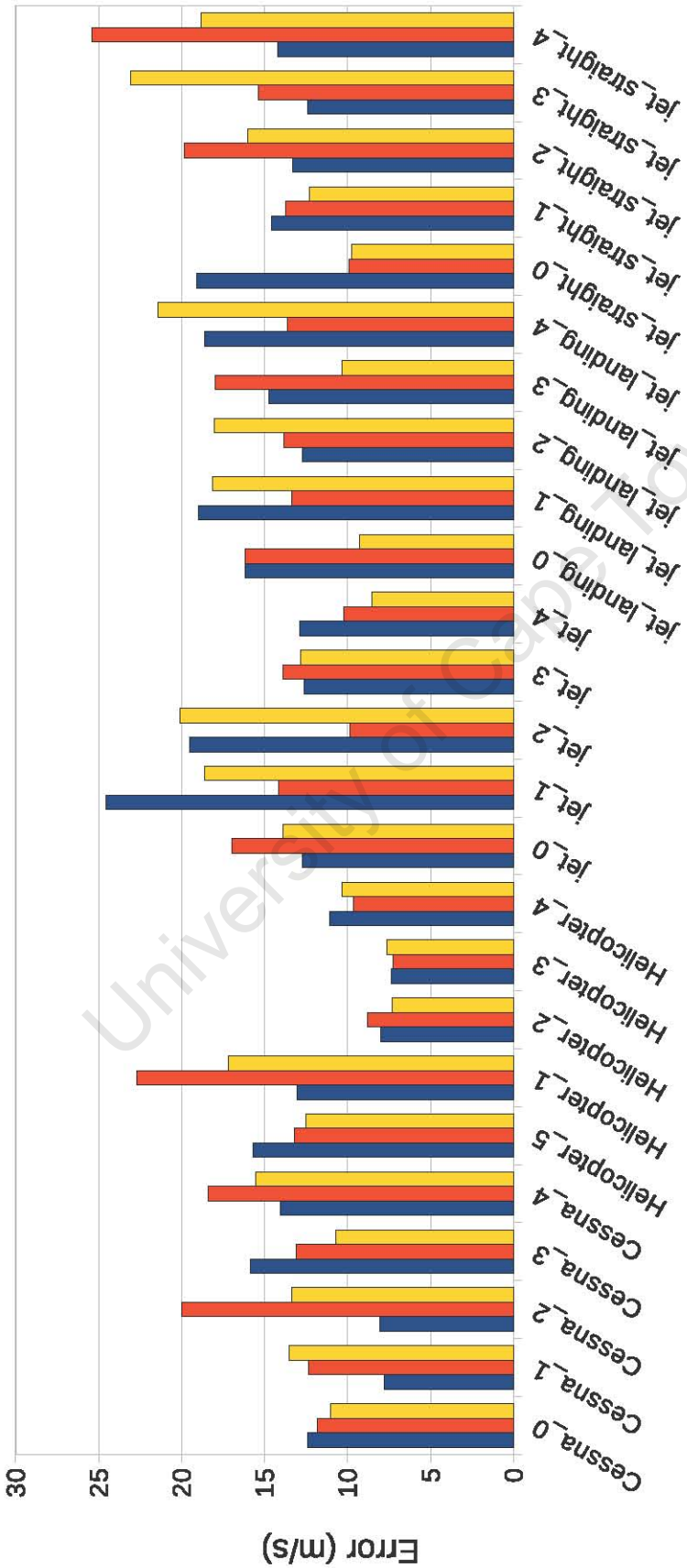


Figure E.4: Velocity estimate errors for altitude

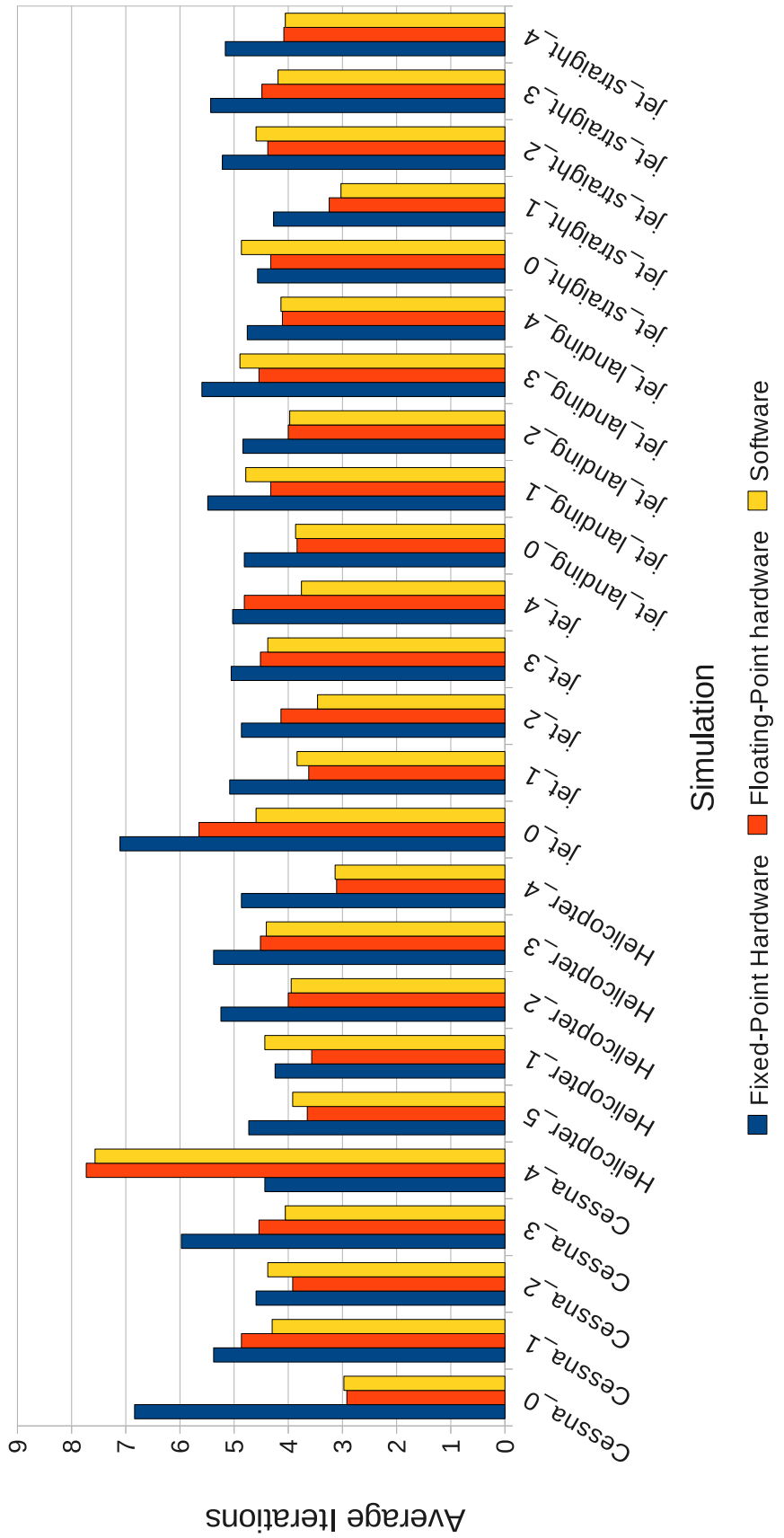


Figure E.5: Average iterations to convergence for the three implementations on all the data sets. Note the slight increase in iterations required for the fixed-point design.

Bibliography

[1] Automatically Tuned Linear Algebra Software website. Online. URL http://math-atlas.sourceforge.net/. Accessed 16 December 2010.

[2] Center for Astronomy Signal Processing and Electronics Research (CASPER). Online. URL http://casper.berkeley.edu/. Accessed 16 December 2010.

[3] NASA Deep Space Network group. Online. URL http://deepspace.jpl.nasa.gov/dsn/. Accessed 16 December 2010.

[4] Interconnect Break-out Board (IBOB) wiki. Online. URL http://casper.berkeley.edu/wiki/IBOB. Accessed 16 December 2010.

[5] MyHDL Python Package for hardware description and verification. Online. URL http://www.myhdl.org/doku.php/start. Accessed 16 December 2010.

[6] PC Magazine. Online. URL http://www.pcmag.com/article2/0,2817,2351266,00.asp. Accessed 16 December 2010.

[7] Reconfigurable Open Architecture Computing Hardware ROACH board wiki. Online. URL http://casper.berkeley.edu/wiki/ROACH. Accessed 16 December 2010.

[8] Xtremedata FPGA-based accelerators. Online. URL http://www.xtremedata.com/products/accelerators. Accessed 16 December 2010.

[9] floating- and fixed-point VHDL-2008 Support Library. Online. URL http://www.vhdl.org/fphdl/. Accessed 16 December 2010.

[10] *NVIDIA CUDA Programming Guide*. www.nvidia.com/cuda, 2.3 edition. Last accessed 16 December 2010.

[11] DRC Reconfigurable Processing Units. Online. URL http://www.drccomputer.com/drc/modules.html. Accessed 16 December 2010.

[12] MATLAB Fixed-Point Toolbox. Online, . URL http://www.mathworks.com/products/fixed/. Accessed 16 December 2010.

[13] MATLAB Simulink Fixed-Point Toolbox. Online, . URL `http://www.mathworks.com/products/simfixed/`. Accessed 16 December 2010.

[14] Nallatech H101-PCIXM Accelerator Card. Online. URL `http://www.nallatech.com/PCI-Express-Cards/h101-pcixm.html`. Accessed 16 December 2010.

[15] Fixed Point Toolbox for GNU Octave. Online. URL `http://wiki.octave.org/wiki.pl?CategoryFixedPoint`. Accessed 16 December 2010.

[16] Fixed Point Library for Python. Online. URL `http://fixedpoint.sourceforge.net/`. Accessed 16 December 2010.

[17] Starbridge Systems. Online. URL `http://www.starbridgesystems.com/`. Accessed 16 December 2010.

[18] Open SysetmC Initiative. Online. URL `http://www.systemc.org/home/`. Accessed 16 December 2010.

[19] A. Suardi A. Geraci G.Ripamonti A.Abba, A. Manenti. Implementation of the Gauss-Newton algorithm for non-linear least-mean-squares fitting in FPGA devices. In *Engineering of Reconfigurable Systems and Algorithms*, 2009.

[20] R. ANDRAKA and R. PHELPS. An FPGA based processor yields a real time high fidelity radar environment simulator. In *Military and Aerospace Applications of Programmable Devices and Technologies Conference*, 1998.

[21] M. Bach, S. Gorbunov, I. Kisel, V. Lindenstruth, and U. Kebschull. FAIR-EXPERIMENTS-38 Porting a Kalman filter based track fit to NVIDIA CUDA.

[22] David Bishop. Fixed- and floating-point packages for VHDL 2005. In *Design and Verification Conference and Exhibition*, 2005.

[23] Robert W. Broderson Chen Chang, John Wawrzynek. BEE2: A High-End Reconfigurable Computing System. *IEEE Design and Test of Computers*, 22:114–125, 2005.

[24] Z. Salcic C.R. Lee. High-performance FPGA-based implementation of Kalman filter. *Microprocessors and Microsystems*, 21:257–265, 1997.

[25] Gildas Genest Craig Steffen. Nallatech IN-Socket FPGA FroNt-SIde BuS accelerator. *Computing in Science and Engineering*, pages 78–82, 2010.

[26] A. Kountzeris M.M. Kharbouch MPhil D.P. Atherton, E. Gul. Tracking multiple targets using parallel processing. *IEE PROCEEDINGS*, 137:225–234, 1990.

[27] D.Phil D.P. Atherton, H.-J. Lin. Parallel implementation of IMM tracking algorithm using transputers. *IEE Prac.-Radar, Sonar Nauig., Vol. 141, No. 6, December 1994*, 141:325–332, 1994.

[28] Julio Villalba Elisardo Antelo and Emilio L. Zapata. Low-Latency Pipelined 2D and 3D CORDIC Processors. *IEEE TRANSACTIONS ON COMPUTERS*, 57:404–417, March 2008.

[29] Bogdan Pasca Florent de Dinechin. the FloPoCo Project. Online. URL `http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`. Accessed 16 December 2010.

[30] Cristian Klein Florent de Dinechin and Bogdan Pasca. Generating High-Performance Custom Floating-Point Pipelines. In *Dans Proceedings of the 19th International Conference on Field Programmable Logic and Applications - Field Programmable Logic and Applications, Prague : Czech Republic*, 2009.

[31] Octavian Cret Radu Tudoran Florent de Dinechin, Jeremie Detrey. When FPGAs are better at floating-point than microprocessors. In *16th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'08)*, page 260, Monterey, CA, USA, 2008. ACM Press.

[32] Octavian CretÂ¸ Radu Tudoran Florent de Dinechin, Bogdan Pasca. An Fpga-Specific Approach To Floating-Point Accumulation And Sum-Of-Products. In *International Conference on Field-Programmable Technology*, 2008.

[33] David Berger Gary Rubin, Earl Sager. GPU Acceleration of SAR/ISAR Imaging Algorithms,. In *The 32nd Annual Symposium of the Antenna Measurement Techniques Association*, 2010.

[34] Robin Bruce Gildas Genest, Richard Chamberlain. Programming an FPGA-based Super Computer Using a C-to-VHDL Compiler: DIME-C. In *Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 280–286, 2007.

[35] Maya Gokhale, Janette Frigo, Christine Ahrens, and Ron Minnich. Monte carlo radiative heat transfer simulation on a reconfigurable computer. In *in Proc. FPL, LNCS 3203*, pages 95–104, 2004.

[36] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23:5–47, 1991.

[37] A. Hartstein and T. R. Puzak. The Optimum Pipe-line Depth for a Microprocessor. In *International Symposium on Computer Architecture*, pages 7–13, 2002.

[38] Robert W. Brodersen Hayden Kwok-Hay So. Improving Usability of FPGA-Based Reconfigurable Computers through Operating System Support. In *16th International Conference on Field Programmable Logic and Applications*, 2006.

[39] Tom Hill. Floating- to Fixed-Point Conversion of MATLAB Algorithms Targeting FPGAs. *Xilinx DSP Magazine*, 2:32–35, May 2006.

[40] IEEE. IEEE Standard for Floating-Point Arithmetic. Online, 2008.

[41] D.Patterson J. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 2007.

[42] Ju-wook Jang, Seonil Choi, and Viktor Prasanna. Energy-Efficient Matrix Multiplication on FPGAs. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, volume 2438 of *Lecture Notes in Computer Science*, pages 373–384. Springer Berlin / Heidelberg, 2002.

[43] Florent de Dinechin Jeremie Detrey. A VHDL Library of Parametrisable Floating-Point and LNS Operators for FPGA. Online. URL `http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/`. Accessed 16 December 2010.

[44] et al K.Asanovic. The landscape of parallel computing research: a view from Berkeley. Technical report, University of California at Berkeley, 2006.

[45] Scott Hauck Katherine Compton. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34:171 – 210, 2002.

[46] McMillan S. Monn G. Schoonover K. Stivers F. Underwood K.D. Ligon, W.B. A re-evaluation of the practicality of floating-point operations on FPGAs. In *Symposium on FPGAs for Custom Computing Machines*, 1998.

[47] C. Dick M. Karkooti, J. Cavallaro. FPGA Implementation of Matrix Inversion Using QRD-RLS Algorithm. In *Proc. 39th Asilomar Conference on Signals, Systems, and Computers*, 2005.

[48] H. Nies M. Lambers, A. Kolb and M. Kalkuhl. GPUbased framework for interactive visualization of SAR data,. In *Proc. Int. Geoscience and Remote Sensing Symposium*, 2007.

[49] J. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Comittee on Computer Architecture (TCAA) Newsletter*, December:19–25, 1995.

[50] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 2002 international symposium on low power electronics and design*, 2002.

[51] Jan Schier Milan Tichy and David Gregg. FPGA implementation of adaptive filters based on GSFAP using log arithmetic. In *IEEE Workshop on Signal Processing Systems Design and Implementation*, pages 342 – 347, 2006.

[52] Joseph Milburn. Co-processor Offloading Applied to Passive Coherent Location with Doppler and Bearing Data. Masters thesis dissertation, University of Cape Town, February 2010.

[53] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38:4, 1965.

[54] Norman Morrison. Gauss-Newton Smoothing for Radar and Tracking. Unpublished, 2009.

[55] Norman Morrison. *Introduction to Sequential Smoothing and Prediction*. McGraw-Hill, 1969.

[56] VPS Naidu and G Girija. Implementation of IMMPDAF Algorithm in LabVIEW for Multi Sensor Single Target Tracking. In *Second International Conference on Cognition and Recognition*, 2008.

[57] Intel Software Network. Intel VTune Performance Analyzer. Online. URL `http://software.intel.com/en-us/intel-vtune/`. Accessed 27 September 2010.

[58] Jean-Michel Muller Nicolas Brisebarre, Florent de Dinechin. Integer and floating-point constant multipliers for FPGAs. In *Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors*, 2008.

[59] Xilinx Technical Staff. CORE Generator Guide. Technical report, Xilinx, Inc, 2003.

[60] Xilinx Technical Staff. CORDIC v5.0 Product Specification. Technical report, Xilinx, Inc., 2009.

[61] Xilinx Technical Staff. Adder/Subtracter v11.0. Technical report, Xilinx, Inc., 2009.

[62] Xilinx Technical Staff. Floating-Point Operator v5.0 Product Specification. Technical report, Xilinx, Inc., 2009.

[63] Xilinx Technical Staff. Divider Generator v3.0 Product Specification. Technical report, Xilinx, Inc., June 2009.

[64] Xilinx Technical Staff. LogiCORE IP Multiplier v11.2 Product Specification. Technical report, Xilinx, Inc., September 2009.

[65] Wonyong Sung and Jiyang Kang. Fixed-Point C Language for Digital Signal Processing. *Asilomar Conference on Signals, Systems and Computers*, 0:816, 1995. ISSN 1058-6393. doi: http://doi.ieeecomputersociety.org/10.1109/ACSSC.1995.540814.

[66] Herb Sutter. The Free Lunch Is Over. *C/C++ Users Journal*, 23, 2005. issue 2.

[67] Altera Corporation technical staff. Designing and Using FPGAs for Double-Precision Floating-Point Math. Technical report, Altera Corporation, 2007.

[68] EE Times. Fixed-point math in C. Online. URL `http://www.eetimes.com/discussion/other/4024639/Fixed-point-math-in-C`. Accessed 16 December 2010.

[69] C.N. Krishnana V. Vaidehia. Scheduling the parallel Kalman algorithm onto multiprocessor systems: a genetic approach. *Control Engineering Practice*, 6(2):209–218, 1998.

[70] Jatin Chhugani Michael Deisher Daehyun Kim Anthony D. Nguyen Nadathur Satish Mikhail Smelyanskiy Srinivas Chennupaty Per Hammarlund Ronak Singhal Pradeep Dubey Victor W. Lee, Changkyu Kim. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*, volume 38, June 2010.

[71] Miriam Leeser Xiaojun Wang. VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3:16, 2010.

[72] CHUNG-RUEY LEE ZORAN SALCIC. FPGA-Based Adaptive Tracking Estimation Computer. *Ieee Transactions On Aerospace And Electronic Systems*, 37:699–706, 2001.