

Imperial College London
Department of Electrical and Electronic Engineering

Precision analysis for hardware acceleration of numerical algorithms

David Peter Boland

December 2011

Supervised by George A. Constantinides

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy of Imperial College London
and the Diploma of Imperial College London

Abstract

The precision used in an algorithm affects the error and performance of individual computations, the memory usage, and the potential parallelism for a fixed hardware budget. However, when migrating an algorithm onto hardware, the potential improvements that can be obtained by tuning the precision throughout an algorithm to meet a range or error specification are often overlooked; the major reason is that it is hard to choose a number system which can guarantee any such specification can be met. Instead, the problem is mitigated by opting to use IEEE standard double precision arithmetic so as to be ‘no worse’ than a software implementation. However, the flexibility in the number representation is one of the key factors that can be exploited on reconfigurable hardware such as FPGAs, and hence ignoring this potential significantly limits the performance achievable.

In order to optimise the performance of hardware reliably, we require a method that can tractably calculate tight bounds for the error or range of any variable within an algorithm, but currently only a handful of methods to calculate such bounds exist, and these either sacrifice tightness or tractability, whilst simulation-based methods cannot guarantee the given error estimate. This thesis presents a new method to calculate these bounds, taking into account both input ranges and finite precision effects, which we show to be, in general, tighter in comparison to existing methods; this in turn can be used to tune the hardware to the algorithm specifications.

We demonstrate the use of this software to optimise hardware for various algorithms to accelerate the solution of a system of linear equations, which forms the basis of many problems in engineering and science, and show that significant performance gains can be obtained by using this new approach in conjunction with more traditional hardware optimisations.

Acknowledgements

After so many years, it is hard to think of all the people who had a hand in making this thesis what it is. Firstly, I have to thank my parents Peter and Elizabeth, who have encouraged my curiosity since my childhood, put up with all the how's and why's and supported me in everything that I've ever done.

I also have to thank my various sources of escape from the small world of academia; special mentions must be made to certain people. My brother Robert and sister Jennifer, for routinely bringing me back to my childhood. My flatmates Richard Bishop and Hugh Spalding who put up with having a student living with them for so many years. Extra kudos to Richard for generally already having a bottle of wine open by the time I got home. All those in CAS who've dealt with me over the course of this PhD. I shall not list all the names, but without the excessively long tea breaks, donut sessions and Thirsty Thursdays, this thesis obviously wouldn't be what it is. Finally my partner Kate Spall, who not only has helped develop my persuasive skills by debating endlessly over pointless topics, but who has also opened my eyes to an entire world outside of academia.

Of course, last of all I must thank my supervisor George. He has shared his ideas and visions, been a perfect sounding board for my thoughts, and it is with the help of his guidance that this thesis is something of which I am proud.

Contents

1. Introduction	1
1.1. Thesis organisation	3
1.2. Statement of originality	5
1.3. Publications	5
2. Background	7
2.1. Hardware acceleration of the conjugate gradient algorithm	8
2.1.1. The conjugate gradient algorithm	8
2.1.2. Hardware technologies	9
2.1.3. Conjugate gradient on sparse matrices	10
2.1.4. Conjugate gradient on dense matrices	14
2.1.5. Conjugate gradient on structured sparse matrices	16
2.1.6. Improving performance by moving away from IEEE 754 single and double pre- cision	18
2.1.7. Summary	22
2.2. Optimising numerical precision within hardware	24
2.2.1. Number systems	24
2.2.2. Tools for range and precision analysis	27
2.2.3. Word-length optimisation	39
2.2.4. Summary	42
2.3. Background summary	43

3. Accelerating iterative algorithms using hardware	45
3.1. Solving a system of linear equations	46
3.1.1. The MINRES algorithm	48
3.1.2. Hardware implementations for solving a system of linear equations	48
3.2. Accelerating the MINRES algorithm using an FPGA	51
3.2.1. Implementation results	59
3.3. Optimising memory bandwidth usage and performance for matrix-vector multiplication in iterative methods	63
3.3.1. Performing matrix-vector multiplication	64
3.3.2. Trading performance with slices	70
3.3.3. Maximising matrix-vector performance	73
3.3.4. Results	78
3.4. Examining the effects of varying the precision used throughout an implementation of the MINRES algorithm	82
3.4.1. Empirical data on the MINRES algorithm in finite precision	82
3.4.2. Estimating the performance impact of varying the precision for a hardware ac- celeration of MINRES algorithm	85
3.5. Summary	87
4. Bounding variable values and round-off effects using Handelman representations	89
4.1. Notation	91
4.2. Creating a polynomial representation of potential range	91
4.2.1. Floating point	92
4.2.2. Fixed point	93
4.3. Calculating bounds of a multivariate polynomial	94
4.3.1. Bernstein expansion	94
4.3.2. Lagrangian duality	96
4.3.3. Theorems of alternatives	97

4.4.	Our heuristic to bound multivariate polynomials	103
4.4.1.	Generalised version of Handelman representations	105
4.4.2.	Algorithm termination	109
4.4.3.	Division	111
4.5.	Testing methodology	112
4.6.	Results	116
4.6.1.	Analysis of our approach on an iteration of the 2x2 conjugate gradient algorithm	116
4.6.2.	Range analysis vs other works	120
4.6.3.	Choosing the maximum number of cancellation terms	128
4.6.4.	Scalability of our approach	130
4.7.	Summary	132
5.	A scalable precision analysis technique	134
5.1.	Scalability of existing approaches	135
5.2.	Controlling the execution time to bound the result of an operation	138
5.3.	Creating a scalable model bounding the range of variable in finite precision arithmetic . .	140
5.3.1.	Representing the range of a variable	140
5.3.2.	Bounding the range of variables in finite precision arithmetic for a user algorithm	142
5.4.	Results	147
5.4.1.	Benchmarks	147
5.4.2.	Test 1: Successive over relaxation	148
5.4.3.	Test 2: MINRES	157
5.5.	Summary	160
6.	Conclusion	161
6.1.	Future work	164
6.1.1.	Improving the tightness of bounds and execution time to compute bounds	164
6.1.2.	Handling loops and conditional statements	165
6.1.3.	Automatically creating optimised hardware designs	165

Contents

6.2. Final remarks	166
A. Example of our heuristic to find Handelman representations for a simple polynomial	167
Bibliography	187

List of Tables

2.1. Comparison of floating point linear solution methods.	24
2.2. Bounds of x/x	38
3.1. Comparison of floating point matrix inversion methods.	51
3.2. Comparison of floating point matrix-vector multiplication in hardware.	64
4.1. Construction of polynomials with floating point error.	93
4.2. Construction of polynomials with fixed point error.	94
4.3. Using linear programming to compute Handelman representations to find bounds of $f(\delta_1) = \delta_1^2$, where $ \delta_1 < 1$	102
4.4. Example polynomials of the form (4.27) to cancel the monomial $-x^2y^2\delta_1^2\delta_2$ from f . . .	108
4.5. Resource usage, max frequency and latency of conjugate gradient implementations. . .	118
4.6. Comparison of execution times to compute range of d_1 and relative error of r_1 for a given precision.	118
4.7. Comparison of our approach vs AA using Chebyshev approximations approach [158]. . .	123
4.8. Comparison of our approach vs SAT modulo theory approach [83, 84].	124
4.9. Execution time of our approach and number of monomials for AA benchmarks [158]. . .	126
4.10. Execution time of our approach and number of monomials for SMT Benchmarks [83]. .	127
5.1. Potential contribution of each monomial in $(1 + x_1)(1 + y_1)(1 + \delta_1)$	139
5.2. Controlling polynomial size using the algorithm defined in Figure 5.1 to control poly- nomial size with $N = 6$	141

List of Tables

5.3. Controlling polynomial size using the algorithm defined in Figure 5.1 with $N = 3$ to control separate polynomials for range in infinite precision and the additional monomials resulting from finite precision errors.	142
5.4. Comparison of execution times to compute average relative error of x variables for a given precision.	152
5.5. Resource usage and max frequency of 5x5 successive over relaxation.	153
5.6. Slice use and max frequency of MINRES implementation required according to analytical tools to guarantee the relative error is less than 1×10^{-3} , or using IEEE standards. .	159

List of Figures

2.1. Conjugate gradient algorithm [97].	9
2.2. Sparse matrix-vector accelerator. Figure taken from [111].	11
2.3. Partial matrix-vector multiplication circuit. Figures taken from [161].	13
2.4. Reduction circuits. Figures taken from [160, 161].	15
2.5. Fully parallel dot-product circuit. Figure taken from [95].	16
2.6. Floating point operators, highlighting additional hardware to implement normalisation and renormalisation. Figure taken from [98].	19
2.7. Comparison of the reduction in norm of the residual for a matrix of order $N = 30$ over iterations between finite (IEEE standard double precision, solid line) and infinite Precision (dashed line) [106].	21
2.8. Number line for a 5 bit unsigned fixed point representation with 2 fractional bits.	25
2.9. Number line for an unsigned floating point representation where the exponent is 2 bits over the range $0 \leq e \leq 3$ and mantissa is 3 bits.	26
2.10. Plot of function $f(x) = 4x - x^2$	30
2.11. Approximations of $1/x$, $x \in [0.25; 1]$. The approximating function is written below the plot, where each variable $\delta_i \in [-1; 1]$	37
3.1. MINRES Algorithm [55].	49
3.2. Circuit data flow.	52
3.3. Plot of pipeline depth for increasing matrix order. This is the minimum number of problems required for the $V^T V$ circuit to always be in full operation.	57

List of Figures

3.4. Plot of percentage efficiency for increasing matrix order using the pipeline depth (3.3). . .	58
3.5. Plot of percentage resource usage on a Virtex 5 LX 330T for increasing matrix order. . .	60
3.6. Comparison of hardware and software performance.	62
3.7. Dot Product Circuit.	65
3.8. Methods to store a banded matrix. Each column will be stored in a separate RAM, as shown in Figures 3.7 and 3.13.	65
3.9. Required multiplication over time. In this figure, the values in grey represent the required vector elements, whilst the values in white represent the required matrix elements from RAM. Any 0 value refers to a multiplication that need not be performed.	66
3.10. Banded dot product circuit for thin bands.	66
3.11. Wrapped wide bands.	67
3.12. Required multiplication over time for wide bands. In this figure, the values in grey represent the required vector elements, whilst the values in white represent the required matrix elements from RAM. Any 0 value refers to a multiplication that need not be performed.	68
3.13. Banded dot product circuits.	68
3.14. Using Delays to Emulate Symmetry.	69
3.15. $\alpha = 2$ Parallel Dot Product Circuits of Bandsizes 4.	72
3.16. $\alpha = 3$ Parallel Dot Product Circuits of Bandsizes 4.	72
3.17. Example circuit for performing dot products in stages, with $\beta = 2$	73
3.18. Maximising performance using ILP.	75
3.19. RAM use.	79
3.20. Maximum performance achievable.	81
3.21. Reduction in residual over iterations for different matrices using various precisions. . . .	84
3.22. Reduction in residual for random matrix of size 100 for various precisions using restarting.	85

List of Figures

3.23. Estimated potential performance using ILP for MINRES. These figures are based upon a circuit running at 150 MHz, which we believe to be a pessimistic operating frequency, and below that used in Section 3.2, but one which all implementations should be able to achieve.	88
4.1. Convex hull property of Bernstein coefficients. Figure taken from [59].	95
4.2. Cancellation algorithm.	110
4.3. Pseudo code for one iteration of the conjugate gradient algorithm on a 2x2 matrix to solve $Ax = b$	113
4.4. Taxonomy of error bounds.	114
4.5. Range and relative error results for various operations in conjugate gradient example. . .	117
4.6. Growth in relative error throughout a conjugate gradient iteration. Operations correspond to pseudo code in Figure 4.3.	121
4.7. Range comparison against published methods.	122
4.8. Graphs illustrating the effect of changing the number of cancellation polynomials ('m'). .	129
4.9. Execution Time growth with number of monomials.	131
4.10. Pseudo code to calculate the product of vector elements.	132
5.1. Algorithm to control the size of the polynomial.	138
5.2. Overall algorithm to find bounds on the range or relative error of a variable from a user input code.	143
5.3. Algorithm to create rational functions bounding intermediate variables.	145
5.4. Algorithm to normalise coefficients.	146
5.5. 5x5 Successive over relaxation benchmark.	148
5.6. MINRES algorithm benchmark.	149
5.7. Range and relative error of various methods to bound error applied to a 5x5 successive over relaxation.	151
5.8. Range and relative error of various methods to calculate bounds applied to a successive over relaxation of a 5x5 matrix.	154

List of Figures

5.9. Trade off between bound on relative error and execution time of various implementations of our approach to find the average bound on the relative error of x vector of a successive over relaxation of a 5x5 matrix.	155
5.10. Trade off between bound on range and execution time of various implementations of our approach to find the average bound on the range of x vector of a successive over relaxation of a 5x5 matrix for various approaches.	156
5.11. Execution time vs number of operations using various methods to bound range applied to a MINRES algorithm of a 4x4 matrix.	158
5.12. Range bounds using various methods to bound range applied to a MINRES algorithm of a 4x4 matrix.	158
5.13. Average bound on relative error for the x vector of the MINRES algorithm using our approach and Affine Arithmetic.	159

List of Acronyms

AA Affine Arithmetic. See Section 2.2.2.

ASIC Application Specific Integrated Circuit.

BRAM Block RAM. Embedded memory on Xilinx FPGAs.

CDS Compressed Diagonal Storage. Technique to store a banded matrix using as few elements as possible, see Section 3.3.1.

CG Conjugate Gradient algorithm. Iterative algorithm to find the solution to a system of linear equations in cases where the matrix is symmetric positive semidefinite, see Section 2.1.1.

CPU Central Processing Unit.

CSR Compressed Sparse Row. Technique to store a sparse matrix using as few elements as possible, see Section 2.1.3.

DSP Digital Signal Processing core. Hardened cores on FPGA devices for performing arithmetic operations. These are faster and use less silicon area than soft-logic implementations.

DSP48E Digital Signal Processing core. Virtex 5 Specific DSP block.

FLOPs Floating-point Operations Per Second.

FPGA Field-Programmable Gate Array.

GHR Generalised Handelman Representation. See Section 4.4.

GPP General Purpose Processor.

GPU Graphics Processing Unit.

IA Interval Arithmetic. See Section 2.2.2.

ILP Integer Linear Programming.

LUT Lookup Table. Soft logic on FPGAs that can be programmed to perform logical operations.

LTI Linear Time Invariant.

MINRES Minimum Residual algorithm. Iterative algorithm to find the solution to a system of linear equations in cases where the matrix is symmetric, see Section 3.1.1.

SIMD Single Instruction Multiple Data. Type of instruction to use when multiple threads all execute the same instruction on many different data inputs independently. Such instructions are trivially parallelisable.

SDP Semi-definite programming. Convex optimization technique to optimise a linear objective function over a cone of positive semidefinite matrices.

SMT Satisfiability Modulo Theory. See Section 2.2.2.

SPD Symmetric Positive-Definite. Type of matrix which is symmetric and has only positive eigenvalues. This type of matrix is required by the conjugate gradient algorithm, see Section 2.1.1.

TwIR Taylor methods with Interval Remainder bounds. See Section 2.2.2.

Nomenclature

α	Number of parallel dot-product circuits. See Section 3.3.2.
β	Number of stages required to perform a dot-product operation. See Section 3.3.2.
δ^λ	A term, as defined in (4.1).
λ	Vector collecting the exponents to describe a term, as defined in (4.2).
Δ	Constant bounding the round-off error of an operation. This is a function of the size of the mantissa, see (4.5).
δ_i	Variable representing the round-off error of the i^{th} floating point operation of an input code.
$\frac{n+n_\epsilon}{d+d_\epsilon}$	Rational function bounding range of intermediate variable in code. n and d are the numerator and denominator polynomials which contribute to the bounds in infinite precision, n_ϵ and d_ϵ store the additional monomials which result from the introduction of finite precision errors.
γ_{lower}	Lower bound on the variable or function of intent, see Section 4.3.
γ_{upper}	Upper bound on the variable or function of intent, see Section 4.3.
$\hat{\gamma}_{lower}$	Computationally tractable lower bound on the variable or function of intent, see Section 4.3.
$\hat{\gamma}_{upper}$	Computationally tractable upper bound on the variable or function of intent, see Section 4.3.
ι	Integer variable for the number of adders implemented in circuit, used in ILP (Figure 3.18).
κ	Integer variable for the number of multipliers implemented in circuit, used in ILP (Figure 3.18).

Nomenclature

ν_j	Binary indicator variable for whether parallel circuit j is used in ILP (Figure 3.18).
ρ	Maximum order of a polynomial.
ρ_{ij}	Integer variable for number of true dual-port RAMs that are required in the j^{th} RAM column of the i^{th} dot-product circuit to store matrix elements for dot-product circuit, used in ILP (Figure 3.18).
σ_{1ij}	Integer variable for number of simple dual-port RAMs that are required in the j^{th} RAM column of the i^{th} dot-product circuit to store matrix elements for dot-product circuit, used in ILP (Figure 3.18).
σ_{2ij}	Integer variable for number of simple dual-port RAMs that are required in the j^{th} RAM column of the i^{th} dot-product circuit to store delays for symmetrical matrix elements for dot-product circuit, used in ILP (Figure 3.18).
τ_{1ij}	Integer variable for number of shift-registers that are required in the j^{th} RAM column of the i^{th} dot-product circuit to store matrix elements for dot-product circuit, used in ILP (Figure 3.18).
τ_{2ij}	Integer variable for number of shift-registers that are required in the j^{th} RAM column of the i^{th} dot-product circuit to store delays for symmetrical matrix elements for dot-product circuit, used in ILP (Figure 3.18).
ζ_i	Added variable to represent the range of all unwanted monomials during polynomial simplification algorithm, described in Figure 5.1.
B	Constant representing the maximum capacity of the BRAMs on the target device, in terms of the number of words they can store, used in ILP (Figure 3.18).
$c\delta^\lambda$	A monomial - a term multiplied by some real coefficient c .
C_1	Constant representing the latency of a dot-product circuit for the MINRES circuit, see equation (3.2).

Nomenclature

C_2	Constant representing the latency of all floating point operations in MINRES circuit, excluding any dot-product operation, see equation (3.2).
C_i	Constant to center the variable ζ_i , which represent the range of all unwanted monomials during polynomial simplification algorithm, described in Figure 5.1.
c_α	Non-negative constant. See equation (4.18).
D	Constant representing the maximum number of DSPs on the target device, used in ILP (Figure 3.18).
$f(\delta)$	Multivariate polynomial representing the value of a variable.
$h(\delta)$	A ‘cancellation polynomial’, used within our heuristic to generate a Handelman representation, see Section 4.4.
I	Number of MINRES iterations required to converge to a solution, see (3.5a).
I_ρ	An interval which bounds the remaining higher order terms created as the result of an operation applied to Taylor methods with interval remainder bounds.
K_1	Constant representing the number of slices required to create a one cycle delay, used in ILP (Figure 3.18).
K_2	Constant representing the number of slices required to create an adder, used in ILP (Figure 3.18).
K_3	Constant representing the number of slices required to create a multiplier, used in ILP (Figure 3.18).
K_4	Constant representing the number of DSPs required to create an adder, used in ILP (Figure 3.18).
K_5	Constant representing the number of DSPs required to create a multiplier, used in ILP (Figure 3.18).
K_6	Constant representing the number of slices required for all vector operations, excluding the dot-product, in the MINRES circuit, used in ILP. See equation (3.8).

Nomenclature

K_7	Constant representing the number of slices required for all scalar operations in the MINRES circuit, used in ILP. See equation (3.8).
K_8	Constant representing the number of DSPs required for all vector operations, excluding the dot-product, in the MINRES circuit, used in ILP. See equation (3.8).
K_9	Constant representing the number of DSPs required for all scalar operations in the MINRES circuit, used in ILP. See equation (3.8).
k_α	Non-negative polynomial. See equation (4.17).
L	Large Constant, used in big-M formulation in ILP (Figure 3.18).
M	Matrix bandsize: Number of non-zero elements from the main diagonal of a matrix.
m	User defined variable for the maximum number of terms used to create a cancellation polynomial in Section 4.4.
N	Order of matrix.
n	Number of input variables required to create a polynomial representing the range of any variable from an input code.
N_1	Constant to define the number of monomials representing the range in infinite precision to be retained during polynomial simplification, described in Figure 5.2.
N_2	Constant to define the number of monomials resulting from the introduction of finite precision errors to be retained during polynomial simplification, described in Figure 5.2.
N_c	Constant to define the number of monomials to be retained during polynomial simplification algorithm, described in Figure 5.1.
n_o	Number of operations in input code.
n_s	Number of splits when applying interval splitting.
n_{sv}	Number of variables that are split when applying interval splitting.

Nomenclature

P	Number of different problems within pipeline of MINRES circuit, see Section 3.2.
p_{lower_ghr}	A generalised handelman representation providing a certificate of a lower bound.
p_{upper_ghr}	A generalised handelman representation providing a certificate of a upper bound.
R	Constant representing the maximum number of BRAMs on the target device, used in ILP (Figure 3.18).
S	Constant representing the maximum number of slices on the target device, used in ILP (Figure 3.18).
S_c	Compact set of positive linear equalities. See equation (4.18).
T_ρ	A Taylor polynomial, which consists of all the terms that are less than or equal to an order (ρ).
X	User constant for the maximum number of parallel dot-product circuits to search for, used in ILP (Figure 3.18).
Y	User constant for the maximum number of stages to perform a dot-product for, used in ILP (Figure 3.18).
Z	Constant used to steer optimisation goal in ILP (Figure 3.18) towards maximising parallelism or minimising RAM use.

Chapter 1

Introduction

Arithmetic computation has become a fundamental necessity in today's digital society. In order to perform any numerical arithmetic, first it is necessary to choose a suitable number system. Historically, calculations were performed with the assistance of mechanical objects and hence physical size was of importance, for example, the dimensions of an abacus or the number of columns on Charles Babbage's difference engine, and consequently the number system would be chosen according to the desired purpose. However, with the advent of electronic computers, number systems were chosen to suit the underlying hardware architecture, restricting the choice of number system to the available hardware platforms. Unfortunately this meant any programmer would have to understand the intricacies of the specific target hardware platform, so in the 1970s Intel sought to stem the tide of number systems [134], and this led to the advent of the IEEE 754-1985 [77] (and later IEEE 754-2008 [78]) standard. Since this date, the subject of number systems has become somewhat esoteric, with hardware architectures largely being designed according to the IEEE standard and most users in the scientific computing community choosing to adopt either IEEE standard single or double precision floating point.

While the adoption of this standard simplified algorithm design for programmers, it has not removed the need to create numerically robust algorithms and it has often led to a programmer overlooking the potential performance that can be obtained by tuning the precision used throughout an algorithm to meet a range or error specification. Even when adhering to this standard, an increase in precision will imply a substantial decrease in the performance of the hardware. As an example, recent figures for the difference in performance, in terms of peak theoretical floating point operations per second (FLOPs),

between single and double precision is approximately a factor of 1 to 2 for a CPU [90], 9 for a graphics processing unit (GPU) [117] and 14 for the IBM Cell multiprocessor [80]. On top of this performance hit, the memory use doubles when moving from single to double precision, and any data transfer to a hardware accelerator, such as a GPU, will take twice as long. For hardware platforms such as field programmable gate arrays (FPGAs) or application specific integrated circuits (ASICs), the choice of precision will affect many more factors: the silicon area, clock speed, latency, memory use and data transfer. As a result, the flexibility in the number representation is one of the key factors that can be exploited, as highlighted when estimating the amenability of an application to hardware in the high level RAT Methodology [75], and hence restricting choice of number system to the IEEE standard severely limits the potential performance of the device.

Notwithstanding the potential performance benefits, the reconfigurable hardware community has largely focused on exploiting the inherent parallelism within algorithms to achieve significant performance improvements over general purpose processors (GPPs), and with the notable exception of digital signal processing applications [36], most approaches opt to use IEEE standard double precision arithmetic so as to be ‘no worse’ than a software implementation. However, there is no guarantee that this is the case because floating point operations are not associative and any parallelism may affect the order of the operations. Similarly, there is a large amount of literature that obtains performance improvements by moving from double to single precision, examples include [3, 132], but rarely is this move accompanied by a proof of the numerical stability. Ideally, one requires an automated method to calculate a proof of numerical stability when using single precision for any given algorithm to ensure it would still work in practice, but any such method could similarly be used to calculate a proof for the *minimum* precision required to meet a given accuracy. As such, we believe any comparison should be whether one can guarantee a given design criterion can be met, as opposed to implementing the equivalent operations of software in hardware; such a design criterion may be an error or range specification so as to ensure convergence or a given safety margin.

The reason that, to date, such comparisons are rarely made is that there is no known method that can calculate tight bounds for the error or range of any variable within an algorithm, given that they are affected by both input ranges and finite precision effects. Finite precision effects are the errors that

occur due to rounding, which is often required in order to represent values using the chosen number system. Whilst the error introduced by the rounding of any single value may be small, over the course of an algorithm the accumulation of these errors can cause a significant deviation from the nominal result, and this could impact issues such as the convergence of a computation. Unfortunately, the analytical tools to estimate this error trade quality of bounds for execution time, with the existing methods lying at the extremum of this spectrum, while any simulation-based method cannot guarantee the given error estimate.

In this thesis, we attempt to create a tool capable of calculating bounds on the range or relative error for a variable in an algorithm that are, in general, tighter in comparison to existing methods and provide a much greater control over the tradeoff of execution time for quality of bounds. Such a tool would help the community to perform any performance comparison in terms of meeting a given design criterion, as opposed to implementing the equivalent operations of software in hardware. Furthermore, calculating tighter bounds than existing analytical methods will help create hardware which uses less silicon area and satisfies the same design criterion, while providing some level of control over the execution time tradeoffs is important to ensure that the approach is applicable to both small and large designs, as well as being of interest within optimisation tools which require repeated calculations of bounds to minimise the individual word-lengths within a hardware datapath.

1.1. Thesis organisation

Chapter 2 introduces the reader to the subject of hardware acceleration of algorithms and the potential performance benefits that can be obtained by tuning the precision used throughout any hardware accelerator. It then discusses the existing techniques to calculate bounds on the range or relative error on variables within an algorithm, and the optimisation tools which make use of these techniques to automatically create hardware designs with minimal silicon area that meet a desired design specification.

In the next chapter, we illustrate how to accelerate algorithms using hardware through the use of a case study: the solution to a system of linear equations using the MINRES algorithm. Within this chapter, we adopt a variety of the techniques we discussed within the previous chapter along with some

application specific improvements and optimisation techniques. Altogether, we show these can be used to automatically generate hardware that makes the best use of the available silicon area whilst taking into account important considerations such as efficiency of hardware and I/O bandwidth to create a parameterisable hardware accelerator with high sustained performance in IEEE 754 single precision floating point. We subsequently discuss the impact of precision on this case study in terms of error and potential performance gains to illustrate the need for the development of the tools we create in the remainder of this thesis.

Chapter 4 is where we introduce our new approach to calculate the bounds on the range or relative error within of a variable in an algorithm. Our approach is based upon modelling input ranges and finite precision errors using a multivariate function, and computing bounds on this multivariate function. As such, in this chapter we introduce the notation and models of error that are used throughout this and the subsequent chapter, along with a description of the various relaxation approaches to calculate bounds on a multivariate function, including the theory behind our new approach. We then describe our heuristic based upon this theory to calculate the desired bounds. This chapter also demonstrates how our approach can be used to optimise hardware circuits for small computational kernels, as well as highlighting the limitations preventing it from being applicable to a general problem.

The focus of Chapter 5 is to address the main limitations of the previous chapter, most notably improving the scalability of the approach to ensure it is applicable to real algorithms including any elementary functions. In this chapter, we also highlight that while many of the existing analytical tools may scale to larger problems than the approach we introduced in Chapter 4, their computational complexity mean that they will still struggle to compute useful bounds for real algorithms in a tractable time. We therefore discuss a new approach in this chapter which is not only much more scalable than the existing methods, but also has some level of control over the trade-off of execution time for quality of bounds, and is still capable of finding bounds that are tighter than the existing analytical approaches. We once again demonstrate how this can be used to create hardware designs using less silicon area than existing methods on some simple case studies, including a small instance of the MINRES algorithm allowing us to relate these tools back to our analysis in Chapter 3.

In the final chapter, Chapter 6, we summarise the work and introduce potential avenues for future

research based upon the findings of this thesis.

1.2. Statement of originality

The three main original contributions of this thesis are each contained within their own dedicated chapter. Within the introduction section of each of these chapters, we discuss the contributions in greater detail; we give a summary of the main contributions here:

- A novel architecture for the acceleration of the MINRES algorithm. This features an illustration of how hardware techniques including parallelism and pipelining can be combined with optimisation tools and application specific considerations to create an efficient and parameterisable hardware accelerator with significant performance improvements over software, as well as a discussion of how precision can affect this performance. (Chapter 3, [11, 12, 14])
- Formulation of a new analytical technique to calculate bounds on the range or relative error on a variable in an algorithm, and the discussion of how this can be used to create hardware designs using less silicon area than can be achieved using existing analytical techniques, whilst meeting the same desired output specification. (Chapter 4, [13, 16])
- The development of a scalable framework to calculate these bounds for much larger algorithms than existing analytical techniques. (Chapter 5, [15])

1.3. Publications

The following publications have been written during the course of this thesis:

- *An FPGA-Based Implementation of the MINRES Algorithm*, David Boland and George A. Constantinides, in Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.379-384, 2008.
- *Optimizing Memory Bandwidth use for Matrix-Vector Multiplication in Iterative Methods*, David Boland and George A. Constantinides, in Proc. Int. Symp. on Applied Reconfigurable Computing

(ARC), pp.169-181, 2010. *Winner of the best paper award*

- *Optimizing Memory Bandwidth Use and Performance for Matrix-Vector Multiplication in Iterative Methods*, David Boland and George A. Constantinides, ACM Transactions on Reconfigurable Technology and Systems, vol. 4, no. 3, pp.22:1–22:14, August 2011.
- *Automated Precision Analysis: A Polynomial Algebraic Approach*, David Boland and George A. Constantinides, in Proc. Int. Symp. on Field-Programmable Custom Computing Machines (FCCM), pp.157-164, 2010
- *Bounding Variable Values and Round-off Effects using Handelman Representations*, David Boland, George A. Constantinides, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 11, pp. 16911704, November 2011.
- *A Scalable Approach for Automated Precision Analysis*, David Boland, George A. Constantinides, to appear in Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA), 2012.

Chapter 2

Background

The focus of this thesis is the development of automated numerical analysis techniques to accelerate hardware; in this chapter we discuss related work to help place our work in context. To this end, we first provide a broad overview of techniques used to accelerate algorithms with hardware. Within this discussion, we highlight that numerical aspects are an important factor to consider when trying to maximise the performance of any custom accelerator for a fixed silicon budget. We subsequently explore the existing tools and techniques available to optimise hardware with respect to numerical precision.

When discussing techniques used to accelerate algorithms with hardware, in Section 2.1 we restrict our focus to a single problem domain: the solution to a system of linear equations $Ax = b$ using the conjugate gradient algorithm. This particular field is of interest because the solution of a system of linear equations forms the basis of many problems in engineering and science and there are many algorithms to find this solution which are very amenable to modern hardware architectures. As a result, this field has attracted a large number of publications regarding hardware acceleration, and consequently will encompass many of the techniques used throughout hardware accelerators. This section is also of interest for comparison purposes against the work we present in Chapter 3, which presents a case study of the use of hardware to accelerate the solution to a system of linear equations and adopts many of the techniques we discuss in this background chapter.

When discussing tools to optimise the precision used in an accelerator, we first describe fixed and floating point number systems to illustrate the source of numerical error, we then analyse the techniques which can be used to perform automated numerical analysis in Section 2.2.1 before we discuss the higher

level tools which make use of these techniques for wordlength optimisation for an implementation of a hardware accelerator in Section 2.2.2. While the work which we present in Chapters 4 and 5 of this thesis can be seen as an alternative to the numerical analysis techniques, we believe it is important to consider how the techniques we develop could be incorporated into existing work to develop high level tools which could make the best use of the hardware available.

2.1. Hardware acceleration of the conjugate gradient algorithm

2.1.1. The conjugate gradient algorithm

In this section we analyse hardware accelerations of the conjugate gradient (CG) algorithm [62]. This is an iterative method to find the solution to a system of linear equations, with the pseudo code for this algorithm given in Figure 2.1. The reason we have chosen this algorithm as a case study is because the algorithm is very well suited to custom hardware accelerators. Firstly, this is because no complex operators, such as a square root, are required; with reference to Figure 2.1, the CG algorithm only consists of addition, subtraction, multiplication and division operations. Furthermore, because most of these are vector-based operations, which we have highlighted using bold font in Figure 2.1, the algorithm is very compute intensive, and this means there is a large amount of room to obtain performance improvements through parallelisation of these operators. Figure 2.1 also highlights that the only conditional operation is checking whether the output criterion is satisfied, meaning there are no complex control overheads with this algorithm. Finally, it also illustrates that there is little data transfer: only the input matrix A and vectors b and x need to be loaded onto the accelerator, while there are potentially many operations in the internal loop, most notably those originating from the repeated matrix-vector multiplication.

The conjugate gradient algorithm is also of interest because precision has a large impact on the convergence of the algorithm, as we will discuss in Section 2.1.6, meaning that performance benefits of reducing precision could be countered by a slower convergence rate. Altogether, this means that the goal of maximising the performance of the conjugate gradient algorithm is a good case study to highlight the trade-offs between precision and finite precision errors. Finally, this algorithm is of interest simply because of the large amount of literature on various hardware accelerations of this algorithm, a

```

Input : Matrix  $A$ , Vector  $b$ , error tolerance  $\varepsilon_{cg}$ 
Output :  $x$  Such that  $\|Ax - b\|_2 \leq \varepsilon_{cg}\|b\|_2$ 
// In this figure, vector variables are represented using bold font

 $d_{cg} = b$ 
 $r_{cg} = b$ 
 $\delta_{cg.orig} = r_{cg}^T r_{cg}$ 
 $\delta_{cg.1} = \delta_{cg.orig}$ 
do
     $q_{cg} = Ad_{cg}$ 
     $\alpha_{cg} = \frac{\delta_{cg.1}}{d_{cg}^T q_{cg}}$ 
     $x = x + \alpha_{cg}d$ 
     $r_{cg} = r_{cg} - \alpha_{cg}d_{cg}$ 
     $\delta_{cg.0} = \delta_{cg.1}$ 
     $\delta_{cg.1} = r_{cg}^T r_{cg}$ 
     $\beta_{cg} = \frac{\delta_{cg.1}}{\delta_{cg.0}}$ 
     $d_{cg} = r_{cg} + \beta_{cg}d$ 
while  $\delta_{new} > \varepsilon_{cg}^2 \delta_{cg.orig}$ 

```

Figure 2.1.: Conjugate gradient algorithm [97].

result of the factors we have just described, meaning that it is a good case study to examine the general techniques used in hardware accelerators.

2.1.2. Hardware technologies

The main forms of hardware that can be used for general computation are Central Processing Units (CPUs), Graphics Processors (GPUs) and Field Programmable Gate Arrays (FPGAs). CPUs contain instruction sets allowing them to perform multiple instructions and are designed to implement a set of sequential instructions at maximum speed. The performance of CPUs, which has been growing with Moore's law, as well as their ease of programming have seen them becoming the most commonly used form of computational hardware. However, though the ability to perform multiple instructions is useful in general, for specific applications this architecture is not necessarily ideal as it has limited parallelism. In recent years, this limit in parallelism is being countered by the trend of increasing the number of cores, but the level of parallelism available in CPUs is still significantly less than in competing reconfigurable technology [21]. Furthermore, as this trend is relatively new, the problem of how to program these

processors to make best use of their cores will soon arise [143].

GPUs, whilst traditionally created for manipulating visual graphics, have recently seen that their hardware architecture can be exploited in many other designs. The GPU architecture consists of many dedicated floating point components using a shared memory that can be used in parallel and heavily pipelined. Together this allows for very high floating point performance figures. Furthermore, with developments such as NVidia's CUDA [65], they also have a simple environment to aid parallel design. Unfortunately, the architecture of GPUs means the performance gains are mainly limited to applications which consist of many threads performing the same operations (SIMD) [119].

FPGAs consist of configurable logic blocks and specialized components such as multipliers/BRAMS or even soft-core processors, all held together by some sort of programmable interconnect [4, 20]. The configurable logic blocks consist of look-up tables (LUTs) which can be configured to implement any sort of logic or RAM. Altogether, this architecture provides the flexibility to implement an entire customised datapath. This may include customising the precision to trade area or performance for error, where we note that an FPGA can implement any precision directly in hardware, unlike CPUs and GPUs which are largely limited to single and double precision floating point, or customising the chosen degree of parallelism, where the amount of parallelism is limited by both the size of the device and the chosen algorithm. This makes it possible to obtain high performance, often with a significantly lower power consumption. The major problem with FPGAs is that they generally run at a much lower clock frequency than CPUs and GPUs and thus to outperform these technologies, FPGAs must implement a sufficient level of parallelism to combat the lower frequency.

As there is a large amount of parallelism in the conjugate gradient algorithm, and it is not generally SIMD, FPGAs should provide the best reconfigurable hardware platform to explore the improvement available through parallelism; in the rest of this section, we analyse various hardware implementations.

2.1.3. Conjugate gradient on sparse matrices

The work by Morris et al. [109–111] is an implementation of the conjugate gradient for Sparse Matrices on an SRC-6 (a hardware platform which contains two Intel Xeon processors and two Xilinx II-6000 FPGAs). This work identifies that within the conjugate gradient algorithm, the matrix-vector

multiplication occupies the largest computation time. It therefore accelerates this using the two FPGAs, with a major focus on taking advantage of the matrix sparsity - i.e. avoiding multiplications by zero elements.

In order to minimise the number of multiplications by zero, the matrix is stored in a special format - 'Compressed Sparse Row (CSR) format'. This consists of storing all non-zero values of the matrix, an index to the column the value lies in, and an index for when each new row begins. Using this it can match a matrix value with its corresponding vector element and then perform the multiplication. The circuit to match and multiply these elements (F1 in Figure 2.2(a)) is parallelised as much as possible, and the results summed with an adder-tree. Due to the limited I/O bandwidth however, there are not necessarily enough parallel multiplications for an entire row, and therefore a 'partial summation unit' (often known as a 'reduction circuit') was used to sum the portions of the matrix-vector multiplication for each row (F2 in Figure 2.2(a)). Further work from the same author showed that by using a different partial summation unit (Figure 2.2(b)), which both had a better scheduling and reduced BRAM requirements [109, 110], the maximum matrix order and performance could be significantly increased.

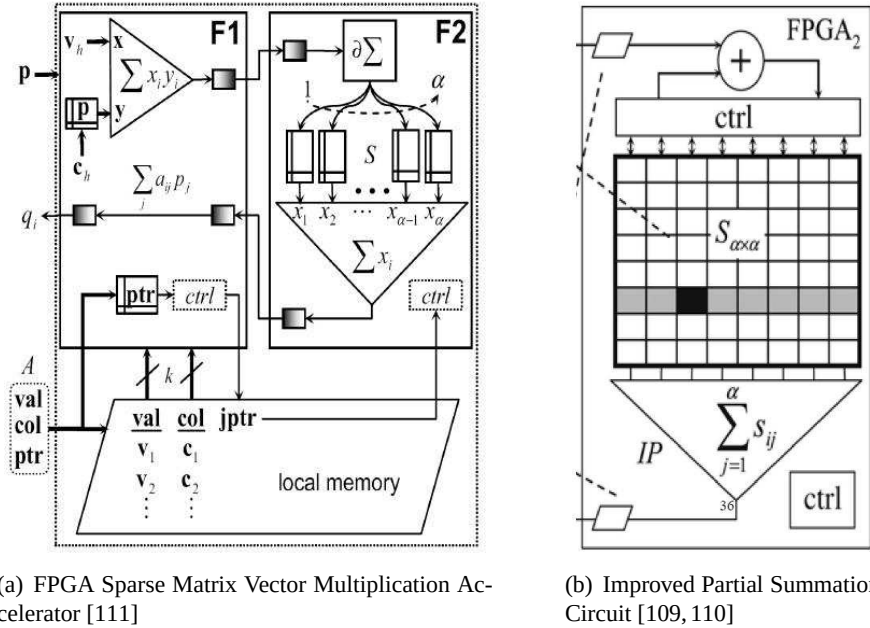


Figure 2.2.: Sparse matrix-vector accelerator. Figure taken from [111].

Sparse matrix vector multiplication (SpMxV)

Whilst the work by Morris et al. implemented the conjugate gradient algorithm on the combined FPGA/Xeon hardware, the FPGA itself only implemented the matrix-vector multiplication. Furthermore, the aim in this work was to obtain the best results using high-level synthesis tools, and it is acknowledged by the author that one may expect to achieve better results if this circuit was designed at the HDL level. Such speculation was subsequently confirmed by a large amount of general research dedicated to Sparse Matrix times Vector (SpMxV) multiplication using FPGAs for SpMxV from several authors. The two main factors that distinguish these approaches are the method to store the matrix and how the on-chip RAM is utilised.

At one extreme is the work by El-Kurdi *et al.* [50, 51], which implements a streaming approach such that the ‘stripes’ containing the non-zeros for the matrix and the vector are held in off-chip RAM and streamed through a set of processing elements, with one processing element for each stripe to achieve the maximum parallelism. The advantage of this approach is that due to the streaming nature, it stores little data on chip, meaning the available on-chip RAM is not an issue and it can operate on arbitrarily large matrices, provided that there is sufficient off-chip RAM. The disadvantage of this approach is that the maximum number of stripes and hence the maximum parallelism is limited by the I/O bandwidth to a 1.76 single precision GFLOPs peak on a Stratix S80 or a sustained 1.5 single precision GFLOPs. In general, streaming approaches are desirable because they are scalable to arbitrary problem sizes, but their potential performance is limited both by the I/O bandwidth and the quality to which they perform any caching of input data.

The work by Morris et al. [109–111] is similar in that it streams the A matrix from off-chip RAM, but by storing or caching the vector on chip, it is possible to achieve a higher performance as it does not need to wait for the correct vector element. Zhuo et al. built on this work [162], the main improvement is that it considers in more detail how to match matrix and vector elements for multiplication. It describes how a basic structure for the partial matrix-vector circuit (Figure 2.3(a)) can be naïve, as at the end of the row one would have to zero pad - adding unnecessary overhead. However, by so-called merging - which involves breaking up this partial matrix-vector multiplier into two (Figure 2.3(b)) and re-using a subtree to perform for a partial multiplication of half the size, the amount of zero padding could be reduced.

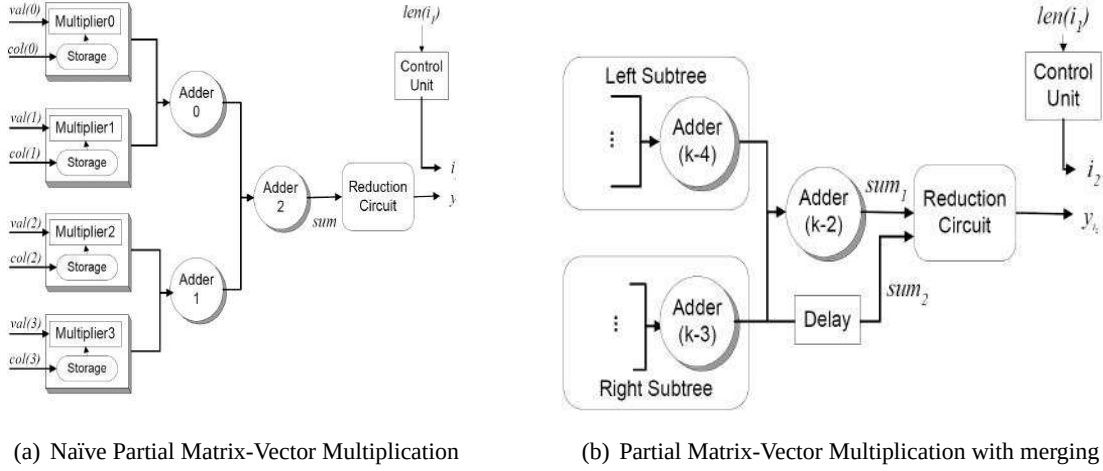


Figure 2.3.: Partial matrix-vector multiplication circuit. Figures taken from [161].

Zhuo et al. [160, 161] went on to explore reduction circuits at HDL level, given that the work by Morris et al. showed that improving the partial summation unit could substantially improve the overall performance. In this work, they consider in detail the problem of attempting to sum serial input data, with the difficulty arising from the fact that floating point adders contain pipelines implying it is not possible to perform a simple loop based accumulator due to data hazards. In these papers, several methods for reduction circuits are proposed (Figure 2.4), with each having different RAM/LUT requirements, and using different scheduling methods to make optimal use of RAM and the adder pipeline to store results. The main result of this work is that it is possible to significantly reduce the circuitry required for a reduction circuit as opposed to a naïve tree based approach. Unfortunately these savings cannot be translated into increased parallelism (by performing dot-products on multiple rows of the matrix simultaneously). This is because the performance of SpMxV was limited by the I/O bandwidth to load the A matrix onto the chip as opposed to the silicon area available on a chip. Indeed, while there are many alternative publications of SpMxV or reduction circuits, the focus of these publications is on clock frequency, silicon area use or ease of implementation. For example, work by Sun et al. [140] shows that matching and multiplying circuits in CSR format and using a single accumulator followed by a reduction circuit similar to Figure 2.4(b) to sum the results inside the pipeline of the accumulator can easily fit on a chip and have a high operating frequency. They also show that it is not necessary to design a specialised reduction circuit for different matrix parameters, but note that performance is still limited

by I/O bandwidth. This problem is unlikely to change in the future FPGAs, because the previous trends have seen the compute capacity grown faster than the I/O bandwidth.

DeLorimier and DeHon [47] acknowledge this problem and create an implementation which similarly targets sparse matrix-vector multiplication for matrices stored in CSR format, but it is specifically aimed at accelerating this function within iterative methods. This operation is special as the same matrix is used for every iteration, as seen in Figure 2.1, and hence this approach suggests loading the matrix into on-chip embedded RAM once, from which it can be re-used multiple times allowing much more parallelism as a result of the significantly higher memory bandwidth. Using this method, it achieved a performance of up to 1.5 sustained double precision GFLOPs on a Virtex2-6000, and showed it is possible to improve the performance by using multiple FPGAs. However, it is also reported that this performance drops as communication time eventually outweighs computation time as the number of FPGAs increases, limiting the peak performance to 12 GFLOPs on 16 FPGAs.

2.1.4. Conjugate gradient on dense matrices

A quick comparison between the work by Morris et al. and DeLorimier and DeHon shows that storing the matrix on chip can lead to a much greater performance, even with the optimised sparse matrix-vector multiplication circuits, such as those suggested by Zhuo et al. This is because the requirements to load this data onto the FPGA creates an I/O bottle-neck (typically determined by the number of off-chip RAMs) limiting the parallelism and potential performance. For example, in the implementation by Morris et al. this limited the number of parallel multiplications of the non-zero matrix values by its associated vector element to just four. However, both works target sparse matrices and use CSR. While this allows them both to target large matrices, the cost of matching matrix and vector elements before multiplication and summing vectors of unknown size significantly reduces the potential performance achievable.

The work by Lopes et al. [95, 97] takes a different approach. It acknowledges that new FPGAs have much larger memories and the floating point support has improved, as indicated by [146]. This work then explores the maximum performance (in terms of FLOPs) that can be achieved on an FPGA implementing the conjugate gradient algorithm on a dense matrix.

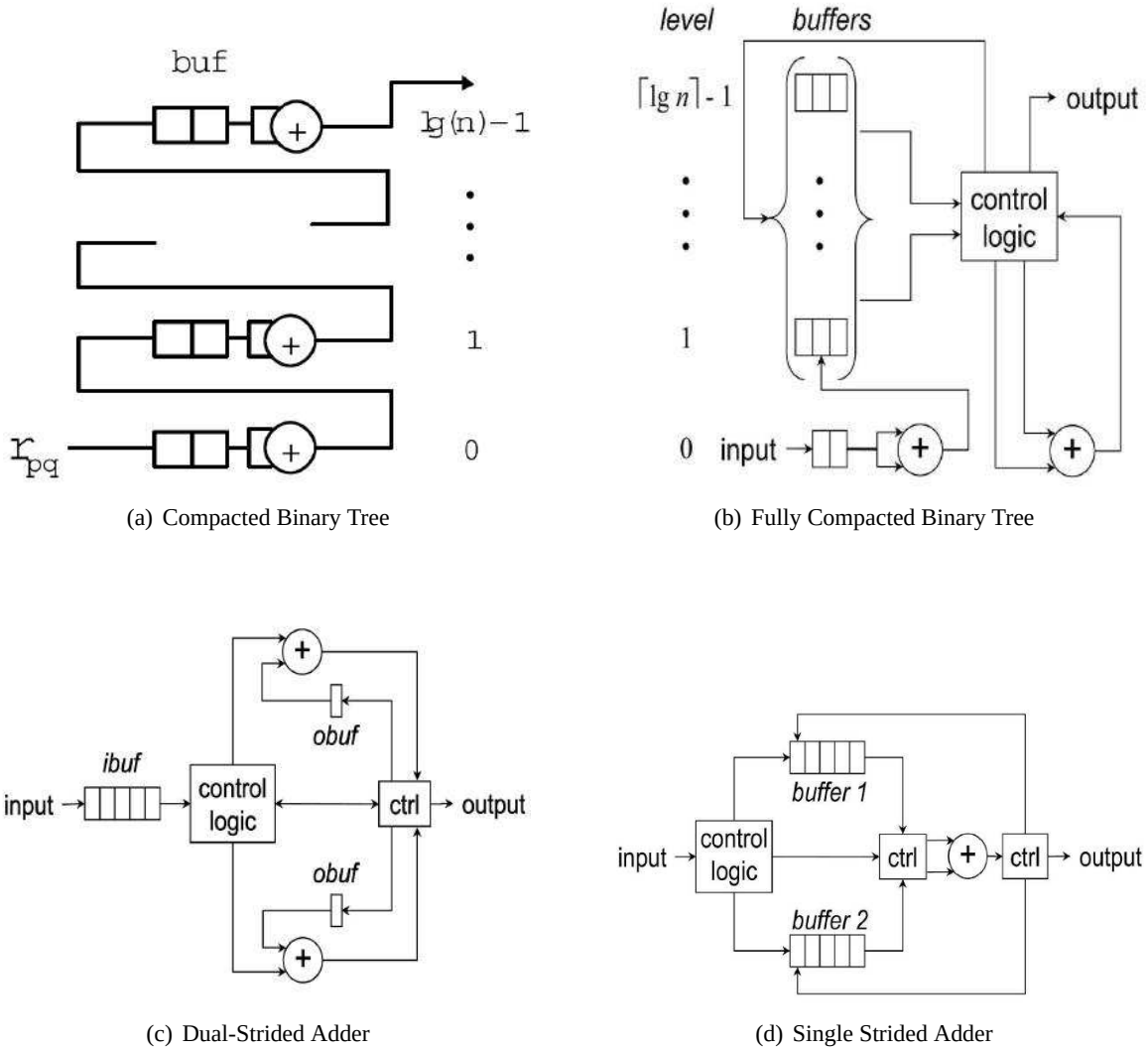


Figure 2.4.: Reduction circuits. Figures taken from [160, 161].

The performance is maximised in two ways, first by having a fully parallelised dot-product core (Figure 2.5) and secondly by ensuring the pipeline remains full through multiplexing several problems into the circuit. Maximising the use of the pipeline ensured the sustained performance remained high, whilst the fully parallel dot-product circuit helped reduce the latency by a factor of N , where N is the order of the matrix, because the matrix-vector multiplication could be performed in the pipeline of this component.

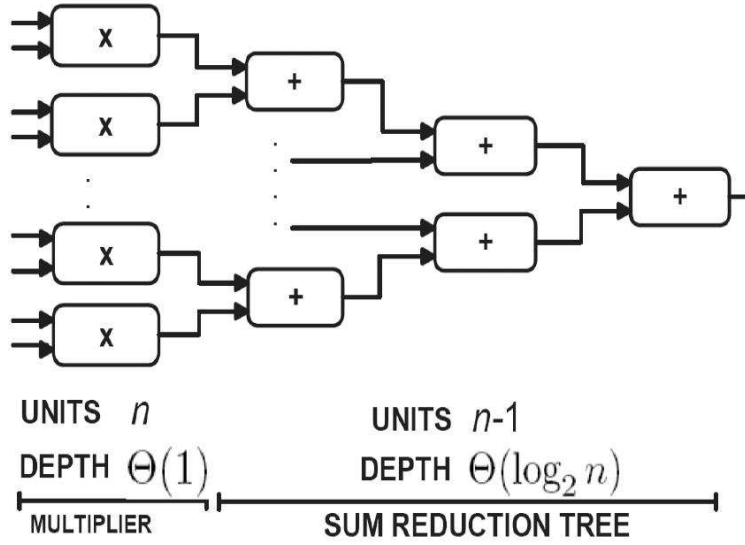


Figure 2.5.: Fully parallel dot-product circuit. Figure taken from [95].

However, the cost of this fully parallel circuit was that in order to ensure the circuit has sufficient data, it must store the matrix on the FPGA itself. Though it was shown the matrix could be loaded whilst solving previous problems, meaning the I/O delay would not impact sustained performance, the BRAM available on the FPGA limited the maximum matrix order.

2.1.5. Conjugate gradient on structured sparse matrices

Comparing the main findings of the work by Lopes et al. with that of Morris et al., the performance of an FPGA implementing the conjugate gradient method could be maximised by storing the matrix on the FPGA as opposed to off-chip, at the cost of the maximum order of the matrix being significantly lower. However, when comparing these two works, it is important to remember that one is a solver for dense matrices and the other for sparse matrices. Given sparse matrix solvers need not operate on or

store any zeros in the matrix, if the FPGA implementation storing the matrix on chip was designed for sparse matrices, one would expect to be able to operate on larger order matrices. A structured sparse matrix can be used to help examine the potential savings of not holding zeros, for the regular matrix structure makes it simpler to implement any parallelism, as opposed to the general sparse solvers using CSR which must match matrix and vector elements on the fly.

Banded sparsity

One example of a structured matrix is a banded matrix. A banded matrix is a structured sparse matrix of the form (2.1), where the non-zero elements are within some known bandwidth m from the main diagonal.

$$\begin{bmatrix} a_{11} & \cdots & a_{1M} & 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \ddots & \vdots & \vdots & \vdots \\ a_{M1} & \vdots & \ddots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \ddots & \vdots & \ddots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots & \ddots & \vdots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \ddots & \vdots & a_{(N-M)N} \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 & a_{N(N-M)} & \cdots & a_{NN} \end{bmatrix} \quad (2.1)$$

Using banded matrices, it is possible to reduce the size of the dot-product circuit to only perform a parallel multiplication the size of the band. This is an approach taken by the author of the dense matrix solver [96], and this shows that the maximum order can be extended to be extended from 92 in the dense case to 236 in the banded case for a thin band size of 5. However this work only implemented a basic architecture which performs parallel multiplication for the size of the band, but stores the entire vector in registers meaning that resource use still grows with matrix order, limiting the potential maximum matrix order of the hardware.

Domain decomposition

A different approach to take advantage of sparsity is to break down a sparse matrix into smaller dense matrices. One approach by Hu et al. [76] uses a domain decomposition method which is designed for finite element matrices to break down a large sparse matrix into 12 independent portions. These portions can be stored on the FPGA and operated on in parallel. Each of these portions require a matrix-vector multiplication which can be done in parallel, each using a reduction circuit. Using this decomposition, this method scaled to very large sparse matrices (orders up to 48000).

2.1.6. Improving performance by moving away from IEEE 754 single and double precision

The main emphasis of the above hardware implementations of the conjugate gradient algorithm have been on maximising the performance in terms of FLOPs or maximising the scalability for single or double precision floating point implementations. However, as FPGAs have much greater flexibility in how to implement floating point operations than CPUs and GPUs, there has been some research into exploiting this flexibility. This section examines several of these techniques.

Fused floating point datapaths

The main reason behind the high performance of the dense or structured sparse designs is that the fully parallel dot product circuit, as seen in Figure 2.5, consists of a large number of floating point operators that are used in parallel and fully pipelined. However, as shown in Figure 2.6, floating point components operate in three stages: normalisation of the input operands, a fixed point operation, and then renormalisation of the result. In a large datapath, such as the dot-product circuit, there will be normalisation and renormalisation at every stage of the accumulation, meaning that removing this hardware could reduce overall latency and save silicon area, which would allow greater room for parallelism or the ability to solve larger problems in the case of structured sparse conjugate gradient problems. This approach is taken by several authors [44, 89, 98].

However, removing the normalisation stages will only provide an equivalent result to that from a circuit consisting of IEEE 754 compliant operators only if it is known *a priori* that no normalisation

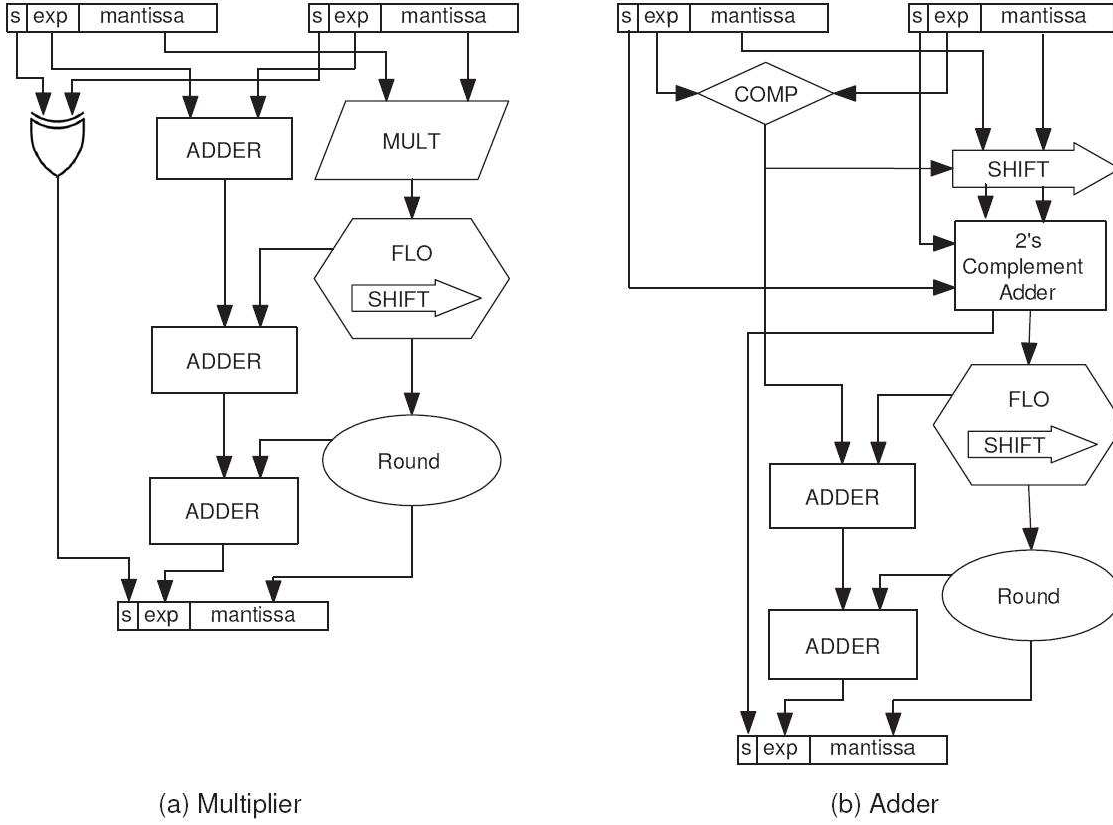


Figure 2.6.: Floating point operators, highlighting additional hardware to implement normalisation and renormalisation. Figure taken from [98].

will be performed. If normalisation is necessary, in order to avoid this error, the precision of the internal fixed point operation must be increased to account for any shifting of mantissas. Choosing the minimum internal precision necessary for a fused datapath involves range analysis, which is one of the problems that is the focus of this thesis. Indeed, this highlights another potential use of the techniques we describe in Chapters 4 and 5. In [44], it is demonstrated that the savings from a pessimistic choice are still significant, while in [98], simulation is used to choose the best implementation, which as we will see in Section 2.2.2, provides no guarantee that the hardware will perform as desired.

Mixed precision solvers

While attempting to emulate single or double precision floating point over a fused datapath can obtain area and performance benefits, it is questionable whether single or double precision FLOPs is indeed

the best method to compare the performance of an accelerator. An alternative metric would be the overall execution time for the hardware accelerator to converge to the correct solution of the system of linear equations. This metric is especially important with iterative algorithms, such as the conjugate gradient algorithm, because the precision used throughout the algorithm can directly affect the rate of convergence.

This reason for this is that in infinite precision, the conjugate gradient algorithm works by constructing an orthogonal basis vector for the so-called Krylov subspace after every iteration, and after N iterations this constitutes the entire subspace from which the true solution can be found. However, when using finite precision, the basis vectors lose orthogonality [64, 106, 121]. The effect this can have on convergence is significant, as seen in Figure 2.7, which shows the decrease in residual over iterations based upon a matrix of size $N = 30$ with large well-separated eigenvalues. The dashed line contains re-orthogonalised vectors (imitating infinite precision) with the solid line representing finite precision (IEEE standard double precision). Clearly the solid line requires many more iterations to reduce the residual significantly.

While this graph implies it is desirable to use as high a precision as possible to achieve the fastest convergence, lower precision hardware can achieve a much higher performance in hardware, and uses less silicon area allowing the potential for much greater parallelism. As a result, there has been some interest in optimising this trade-off, with two main techniques suggested: mixed precision schemes and word-length optimisation.

In mixed precision schemes, the general idea is to perform operations in a lower precision to find an approximate solution, and then use the result as an initial guess to find a more accurate solution in higher precision. Strzodka and Göddeke [139] show that by performing this repeatedly, using a low precision conjugate gradient algorithm in an inner loop to find approximate solutions which are then refined in a high precision outer loop, it is possible to perform most of the operations in lower precision and only a few operations in high precision, albeit with a slightly higher total number of operations, and obtain equivalent final results to an implementation only using high precision. Butarri et al. [22] and Langou et al. [90] showed that doing this on a general purpose processor, with single and double precision as the low and high precisions respectively, can achieve performance speeds approaching those of a sin-

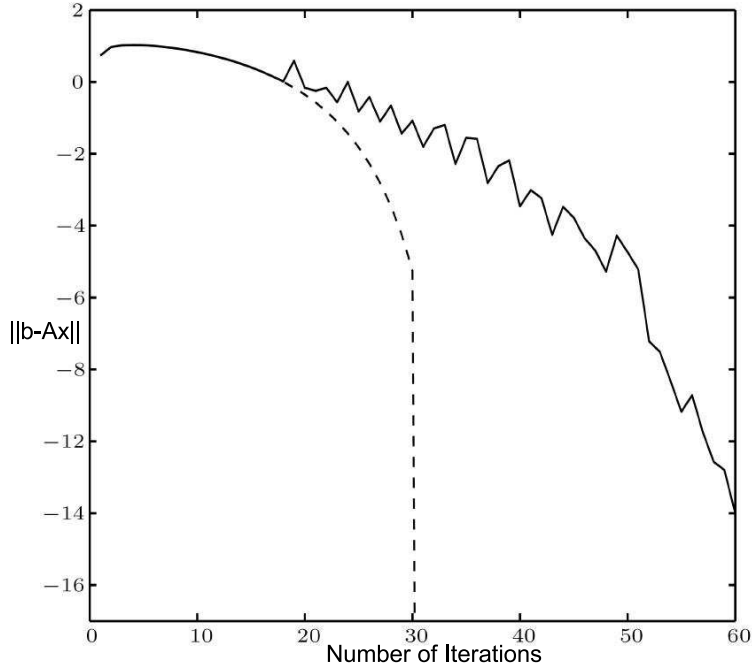


Figure 2.7.: Comparison of the reduction in norm of the residual for a matrix of order $N = 30$ over iterations between finite (IEEE standard double precision, solid line) and infinite Precision (dashed line) [106].

gle precision implementation whilst achieving results equivalent to a double precision implementation. Geddes and Zheng [60] and Sun et al. [141] show similar results for different iterative algorithms, with the latter implementing the lower precision algorithm on an FPGA, and as this runs 2-3 times faster than the equivalent double precision implementation, it is possible to gain a significant overall speed-up. Chow et al. [29] similarly show performance gains of up to $16.7\times$ over a CPU for function comparison.

In word-length optimisation, the aim is simply to choose a single global word-length that optimises the trade-off of performance and number of iterations. Related work includes that by Lopes et al. which examine this trade-off for the conjugate gradient algorithm in [99] and a fused dot-product circuit [98], and Wang et al. [149] who created a reduction circuit based upon the work by Zhuo et al. to investigate the savings that could be achieved by varying precision using a library developed in northeastern university [7], and this showed that the size of the circuit could be reduced proportionally with precision, allowing for much more parallelism.

However, it is important to note that all of these results are based upon simulation, meaning that for

any input to the accelerator, in the case of the mixed precision schemes, there is no guarantee that the lower precision computations will move towards the solution, and in the case of the word-length optimisation schemes, there is no guarantee that the chosen precision is sufficient to achieve convergence. The work we present in this thesis in Chapters 4 and 5 aims to create analytical results to aid such designs, as such, we will discuss word-length optimisation in general in greater detail in Section 2.2.3.

Different number systems

Finally, it is worth noting that further improvements can be obtained by using different number systems than floating point. These include a rational number system used for the conjugate gradient solver in [105], which has been shown to work with significantly larger orders - orders of up to 1024 for a band of size 5 system, despite working with a significantly older FPGA (with much less on chip RAM) and a logarithmic number system for the conjugate gradient method [23]. However, both rely on simulation to choose the number of bits for their number system, and once again, this could create unreliable hardware. The study of different number systems is beyond the scope of this thesis, but it is worth noting that choosing a custom representation using any number system will still rely on some range analysis or relative error analysis to make the best choice of precision; the work we present in this thesis in Chapters 4 and 5 help to provide bounds to aid such choices.

2.1.7. Summary

Overall, this section has highlighted several techniques used throughout the literature to accelerate the conjugate gradient algorithm using hardware. For completeness, given that the above approaches have all been based upon FPGAs (a result of the suitability of the algorithm for this technology as argued in Section 2.1.2), it is important to consider the other technologies. Several of the papers discuss the performance improvement relative to a CPU, but not to GPUs. As a result, it is important to briefly consider GPU implementations. One such implementation is seen by Bolz et al. [17] for sparse conjugate gradient. Whilst details of the work will not be heavily described, it is sufficient to say a large amount of work has been put into mapping the instructions into a SIMD style. Using this approach, the speed-up is relatively low compared to a CPU, and this improvement is smaller than has been shown for FPGAs.

The main performance limitation for the GPU is due to cache thrashing; we note that this also affects CPUs, for which it has been shown that when it is possible to hold most or all of the A matrix in the cache of the processor the performance in software can outperform that of the FPGA [111].

In terms of the FPGA accelerations, the main factors which have been varied are the use of memory, which has consequences on the amount of parallelism, and the choice of precision. With regard to precision, we have discussed how performance gains by varying word-length or through using mixed precision schemes can be obtained, but these improvements have all been based on simulation, and this may result in unsafe hardware, an issue we attempt to address in this thesis. In terms of the memory and performance trade offs, Table 2.1 highlights the main points of interest for each of these implementations to help show the major differences between the approaches. In this table, the main reason for the difference in performance is the choice of whether or not to use off-chip RAM and also whether to operate on sparse matrices. Using off-chip RAM to hold the intermediate results enables larger matrices to be held, but the I/O requirements to load onto the FPGA creates a bottle-neck (typically determined by the number of off-chip RAMs) upon performance and efficiency, typically leading to low speedups, as shown in Table 1 for [46] and [145]. Sparse matrix solvers can handle much larger matrices as it is not necessary to hold or operate on any zeros in the matrix, but representing the matrix using a sparse format (such as CSR) is likely to add significant overhead, and this is unnecessary for dense matrices or matrices with structured sparsity. For these matrices, it is possible to have much simpler datapaths, allowing potentially greater performance. However, to take advantage of this performance, it is necessary to overcome the I/O bandwidth restrictions to load data on-chip; this can be achieved by using the large embedded memories on modern FPGA devices, which now have the capacity to buffer large matrices on chip. However, we acknowledge that the choice of this trade-off will of course depend on application, for example, if the matrix is sparse, though the dense matrix solver would still obtain a high performance in terms of FLOPs, this would include unnecessary multiplications.

We use the discussion in this background as inspiration for our case study in Chapter 3. We first examine how much of the hardware techniques used in the work on accelerating the conjugate gradient algorithm and matrix vector multiplication can be easily generalised in hardware to other iterative algorithms, in our case study, this is the MINRES algorithm. It then expands on issues including how to

Table 2.1.: Comparison of floating point linear solution methods.

Method		Year	GFLOPS	Vs. Software	Max Order	Sparsity	Off-chip RAM	Device
SpMxV	[51]	2006	1.76	$0.5 \times$ vs P4	∞	Sparse	Yes	Stratix S80
SpMxV	[47]	2005	1.5 double	$0.8 \times$ vs P4	Large	Sparse	No	Virtex II
SpMxV	[47]	2005	12 double	$4 \times$ vs P4	Large	Sparse	No	16 $\times$ Virtex II
SpMxV	[162]	2007	2.88	$30 \times$ vs Itanium 2	Large	Sparse	Yes	Virtex II
CG	[105]	2005	0.27 equiv	N/A	1024	Banded	No	Virtex2-Pro
CG	[105]	2005	15 equiv	N/A	3500	Banded	No	Virtex4
CG	[23]	2006	0.91	>1	Large	Sparse	No	Virtex II
CG	[111]	2006	N/A	$1.3 \times$ vs. Xeon	2000	Sparse	Yes	2 $\times$ Virtex II
CG	[111]	2008	N/A	$2.4 \times$ vs. Xeon	Large	Sparse	Yes	2 $\times$ Virtex II
CG	[76]	2008	N/A	$40 \times$	48000	Sparse	No	Virtex 4
CG	[97]	2008	35	$28 \times$ vs. Opteron	58	Dense	No	Virtex5
CG	[96]	2008	N/A	$10 \times$ vs. Opteron	236	Banded	No	Virtex5
CG	[96]	2008	32	N/A	92	Dense	No	Virtex5
CG	[17]	2003	N/A	$2 \times$	Large	Sparse	Yes	GPU

best make use of hardware, design for optimum throughput, determine pipeline depths, and determine exactly how efficiently does the accelerator use the available hardware. We also examine in greater detail the memory and performance trade-off which we have highlighted in this section as the main difference in the performance, attempting to find ways to obtain greater control over this trade-off, as well as ways to maintain high performance to larger matrices, given that this will be the case in many problems in scientific computing. We achieve this by exploring what matrix characteristics can be exploited to further improve the performance and make best use of FPGA resources. Finally, the issues of designing mixed precision solvers serve as inspiration to the work in this thesis in Chapters 4 and 5 which work towards more analytical solutions to aid these designs. We also note that GPUs will require many of the same considerations as FPGAs with regard to making choices between double and single precision, for example, similar analysis for mixed precision schemes have been performed for GPUs [61].

2.2. Optimising numerical precision within hardware

2.2.1. Number systems

In order to optimise the precision in hardware designs, one must choose a number system to ensure the errors as a result of using a given finite precision representation lie within a threshold. To this end, we first introduce the errors that can arise from the use of fixed and floating point number systems, before we discuss the techniques to track the accumulation of errors and prevent any large errors, as well

as the tools which apply these techniques to automatically minimise the silicon area or maximise the performance when designing hardware that meets an end users' design criteria.

In order to highlight the errors that result from the use of finite precision arithmetic, we provide a brief background of binary fixed and floating point number systems. We have restricted this discussion to these two number systems because these are the two most commonly number systems, and as such, our research in this thesis is focused on improving hardware design in these number systems.

Binary fixed point arithmetic

Binary fixed point represents numbers using a fixed number of integer and fractional bits. Negative numbers can either be represented using a two's complement notation, or a sign and magnitude notation. The advantage of a fixed point number system is that the operators can be much more simple than their floating point equivalents, as we have seen in Figure 2.6. The disadvantage is that if the dynamic range of the intermediate variables throughout the course of an algorithm is large, it will require a large number of bits to represent, and this would result in very large hardware.

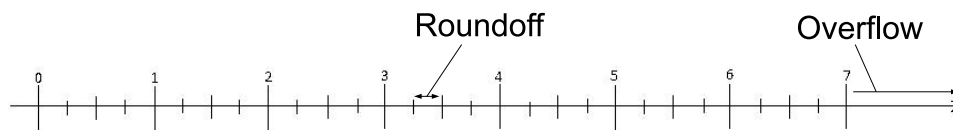


Figure 2.8.: Number line for a 5 bit unsigned fixed point representation with 2 fractional bits.

There are two sources of error in fixed point: overflow and round-off. Overflow occurs when the result of an operation requires more integer bits to represent than are present in the number system. Round-off error occurs when the result of an operation requires more fractional bits to represent than are present in the number system. We highlight these errors using a number line for a 5 bit unsigned fixed point number in Figure 2.8, where the vertical lines are representable numbers, anything to the right of 7 will be an overflow error, and any of the whitespace between vertical lines represents round-off error. While overflow can result in a large difference between the represented result and the true result and must be avoided, the maximum rounding error is less than the least significant bit, meaning it is small and in many cases may be tolerated. However, over the course of an algorithm, rounding errors can accumulate to cause a significant deviation from the true result, and this may affect issues such as the convergence

of an algorithm, as we have seen for the conjugate gradient algorithm in Section 2.1.6.

As a result, when designing a hardware accelerator, the aim is to ensure that overflow does not occur and that the accumulation of errors lies below a threshold stated by the algorithm designer, using the minimum number of bits necessary, to restrict the size of the fixed point operators.

Binary floating point arithmetic

With binary floating point arithmetic, numbers are represented using a sign bit, an m -bit mantissa and a e -bit exponent as shown in (2.2). In contrast to fixed point, the exponent allows this representation to have a much greater dynamic range, the main cost of this is additional hardware to perform normalisation of input operands.

$$x = \pm 2^e \times 0.b_1b_2\dots b_m, b_i \in \{0, 1\} \quad (2.2)$$

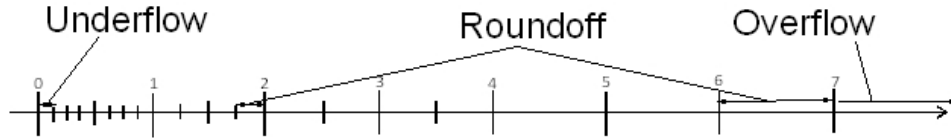


Figure 2.9.: Number line for an unsigned floating point representation where the exponent is 2 bits over the range $0 \leq e \leq 3$ and mantissa is 3 bits.

In floating point as well as overflow and round-off error, there is also underflow error. To illustrate how these errors are different to fixed point errors, a number line expressing the positive numbers for a binary floating point number system with a 2-bit exponent e which lies in the range $0 \leq e \leq 3$, and 3-bit mantissa is shown in Figure 2.9. In this figure, numbers greater than 7 cannot be represented, showing overflow error, somewhat similar to fixed point; to reach larger numbers, a larger exponent is required. The underflow error, which is also a function of the size of the exponent, can be seen where numbers between 0 and 0.125 cannot be represented. The round-off error in floating point is also different to fixed point, because the round-off errors are a function of the magnitude of the nearest representable floating point value. This can be seen in Figure 2.9 where the round-off error for a variable x in this number system is smaller in the region $0.5 \leq x \leq 0.625$ in the region $6 \leq x \leq 7$. This results in the need

to model fixed and floating point rounding errors differently, as we will see in Chapter 4.

As with fixed point, overflow error can potentially be very large and must be avoided. As the maximum underflow error is bounded by the choice of the exponent, generally this will be small, and as such, one must design hardware to ensure the accumulation of round-off and underflow errors lie below a threshold and overflow does not occur.

In order to prevent overflow and to minimise underflow, it is important to perform so called ‘range analysis’ problem which involves ensuring that over the range of input data, there is sufficient dynamic range to prevent overflow. However, when optimising word-lengths, determining the range as a result of the inputs is only a part of the problem; due to rounding errors it is also important to perform ‘precision analysis’, which is typically described as ensuring that the error at the output, caused by the use of a finite precision, lies below a threshold. In this section, we discuss the existing tools and techniques to perform both range and precision analysis.

2.2.2. Tools for range and precision analysis

Simulation

The most straightforward way to estimate an error is through simulation. The aim of any simulation-based approach is to find the inputs which will cause the extreme ranges of the data set. Unfortunately, the size of the search space for the inputs will generally be too large to explore exhaustively, meaning that simulation can only estimate the error because corner cases can be missed. Furthermore, while the quality of the estimate can be improved by increasing the size of the training set or the search time or by using advanced simulation methods, for example statistical profiling [159] or a representative training data set [88,99], in either case, the estimate still does not form a bound. We also note that while it is possible to use methods to avoid precision errors *a posteriori* at run-time, at a cost of execution time [127], our goal is to calculate bounds *a priori* so as to design hardware with the minimum precision.

Interval arithmetic

To calculate true bounds for general algorithms, the most well known analytical approach is interval arithmetic (IA) [108]. Interval arithmetic represents every value as lying within some interval: $[x_1; x_2]$,

where x_1 and x_2 are the lower and upper bounds respectively. The intervals are then propagated through the computation according to basic rules, given in (2.3), which calculate at each stage the new worst case bound.

$$\begin{aligned}
 [x_1; x_2] + [y_1; y_2] &= [x_1 + y_1; x_2 + y_2] \\
 [x_1; x_2] - [y_1; y_2] &= [x_1 - y_2; x_2 - y_1] \\
 [x_1; x_2] \times [y_1; y_2] &= [\min(x_1 y_1, x_2 y_1, x_1 y_2, x_2 y_2); \\
 &\quad \max(x_1 y_1, x_2 y_1, x_1 y_2, x_2 y_2)] \\
 \frac{[x_1; x_2]}{[y_1; y_2]} &= \begin{cases} \text{undefined} & \text{if } 0 \in [y_1; y_2] \\ \left[\min\left(\frac{x_1}{y_1}, \frac{x_1}{y_2}, \frac{x_2}{y_1}, \frac{x_2}{y_2}\right); \right. & \text{otherwise} \\ \left. \max\left(\frac{x_1}{y_1}, \frac{x_1}{y_2}, \frac{x_2}{y_1}, \frac{x_2}{y_2}\right) \right] & \end{cases}
 \end{aligned} \tag{2.3}$$

However, interval arithmetic suffers from the so-called dependency problem, where if the same variable is used twice, information is lost. A trivial example is the following: for a variable x which lies in the interval $[x_1; x_2]$, perform the operation $x - x$. The interval should be $[0, 0]$, but the result using interval arithmetic would be $[x_1 - x_2; x_2 - x_1]$. Several simple examples can demonstrate how this problem may cause bounds that are significantly wider than the tightest bounds [108]. As a result of these problems, there is an active community of researchers in *robust computing* who have developed ways to mitigate this problem, here we outline a variety of techniques that can be used to improve bounds [49].

Creating interval arithmetic friendly algorithms: One approach is to modify the algorithms and create so-called ‘self-validating methods’ which aim to use the original data and minimise the dependencies where possible, instead of the ‘naïve’ approach of directly applying interval arithmetic [49]. For example, if we were to evaluate the function $4x - 4x^2$, where x lies in the interval $[0; 1]$, the ‘naïve’ approach would be evaluated as in (2.4), returning the bounds $x \in [-1; 4]$, but if this were to be re-written as $4 - (x - 2)^2$, as shown in (2.5), this would give the best bounds $x \in [0; 3]$. Unfortunately, whilst such approaches are useful to obtain reliable proofs using IA, the modifications to the algorithm may increase the computational complexity of the algorithm and do not necessarily improve the true numerical stability, instead the modifications simply reduce the sensitivity to which IA bounds this error. Ideally,

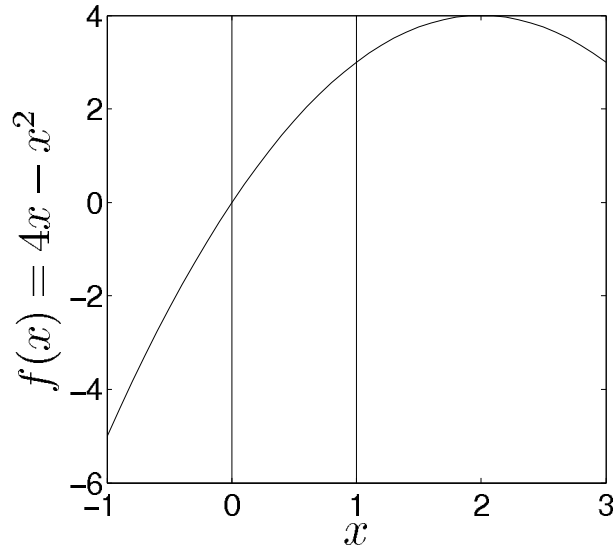
it is preferable to find proofs of tighter bounds for the unmodified algorithm, unless it can be proven that the numerical properties of the modified algorithm have been improved. Furthermore, in general, performing modifications to an algorithm will not be straightforward.

$$\begin{aligned}
 x &\in [0; 1]. \\
 4x &\in [0; 4]. \\
 x^2 &\in [0; 1]. \\
 4x - x^2 &\in [-1; 4].
 \end{aligned} \tag{2.4}$$

$$\begin{aligned}
 x - 2 &\in [-2; -1]. \\
 x - 2^2 &\in [1; 4]. \\
 4 - (x - 2)^2 &\in [0; 3].
 \end{aligned} \tag{2.5}$$

Derivative versions: An alternative technique that can be used to tighten bounds is to use derivative information to improve interval arithmetic. The basic idea is to check if the derivative excludes zero, in which case it is known that the function is monotonically increasing or decreasing and meaning that the true bounds will lie at the extremes [108]. In the case of the above example, if $f(x) = 4x - x^2$, then $f'(x) = 4 - 2x$, and hence because over the interval $x \in [0; 1]$, $\forall_{x \in [0; 1]} f'(x) > 0$, then there are no turning points and the true bounds must lie at the extremes. This is highlighted by Figure 2.10, where the turning point lies outside of the region of interest. As a result, the final bounds can then easily be calculated $[f(0) = 0; f(1) = 3]$. Furthermore, as the bounds of the derivative when computed by interval arithmetic may be wider than the true bounds for the derivative, they could potentially be refined by computing their derivative recursively [104].

The main difficulty with this approach is that computing the differential function either symbolically or numerically via finite differences can be difficult. This has lead to a large amount of research into the

Figure 2.10.: Plot of function $f(x) = 4x - x^2$.

topic of automatic differentiation, where the derivative is computed numerically in conjunction with the regular algorithm execution using a basic set of rules based on algebraic identities and theorems such as the chain rule [129], some basic identities are given in (2.6). The case of the earlier example with the derivative evaluated using automatic differentiation is shown in equation (2.7). We note that in this case, the interval computed by automatic differentiation is equivalent to that calculated symbolically and because the interval of the derivative $([2; 4])$ does not span zero in the final stage the interval could be reduced by re-evaluating the function with the extremes of x .

$$(f, f') + (g, g') = (f + g, f' + g') \quad (2.6)$$

$$(f, f') * (g, g') = (fg, fg' + gf')$$

$$\begin{aligned}
 x &\in ([0; 1], [1; 1]); \\
 4x &\in ([0; 4], [4; 4]); \\
 x^2 &\in ([0; 1], [0; 2]); \\
 4x - x^2 &\in ([-1; 4], [2; 4]);
 \end{aligned} \tag{2.7}$$

However, we do note that if the derivative does include zero over the range of interest, the bounds must be computed using interval arithmetic as usual. For example, for the function $f(x) = x - x^2$, $x \in [0; 1]$, $f'(x) = 1 - 2x$, and hence the range of $f'(x)$ is $[-1; 1]$, meaning that the interval arithmetic must be performed, as in (2.8), returning the bounds $[0; 1]$ when the ideal bounds can be found by re-writing the equation into the form $1/4 - (x - 1/2)^2$ returning the bounds $[0; 1/4]$. This method also suffers in the case of multivariate functions, because it is necessary to compute the partial derivative with respect to every variable, and ensure every partial derivative does not span zero, and computing many partial derivatives means that the execution time of this approach scales in proportion to the number of variables.

$$\begin{aligned}
 x &\in [0; 1]; \\
 x^2 &\in [0; 1]; \\
 x - x^2 &\in [0; 1];
 \end{aligned} \tag{2.8}$$

Interval splitting/subdivision: One simplistic method to reduce the effect of the dependency problem is to split intervals into the union of much smaller intervals (2.9), and evaluate each of these independently because dependencies of smaller intervals have a reduced effect in widening bounds (2.10). This is shown for the example in equation (2.11), which performs the same computation as (2.4), but with the initial interval split into two independent portions, and we note that this approach obtains tighter bounds. However, while effective, the number of intervals that must be evaluated grows $O(n_o n_s^{n_{sv}})$, where n_o is the number of interval operations, n_s is the number of splits, and n_{sv} is the number of variables that are

split.

$$[x_{lower} \ x_{upper}] = \bigcup_{i=1}^n [x_{ilower} \ x_{iupper}]; \quad (2.9)$$

$$f([x_{lower} \ x_{upper}]) \supseteq \bigcup_{i=1}^n f([x_{ilower} \ x_{iupper}]); \quad (2.10)$$

$$\begin{aligned} x &\in [0; 1/2] \bigcup [1/2; 1] \\ 4x &\in [0; 2] \bigcup [2; 4]; \\ x^2 &\in [0; 1/4] \bigcup [1/4; 1]; \\ 4x - x^2 &\in [-1/4; 3/4] \bigcup [-1/4; 15/4]; \end{aligned} \quad (2.11)$$

Global interval methods: As a result of the number of techniques available to refine bounds using interval arithmetic, there exists a large amount of literature on global optimisation frameworks which automatically perform these techniques to improve bounds using interval analysis [66, 69, 123] as well as a set of practical tools which we will briefly analyse here.

The most general tool, INTLAB toolbox [131], provides an optimised toolbox for Matlab which can perform all of the techniques listed, but unfortunately it requires a user to choose the techniques to refine the bounds. Of greater interest is the work by Kinsman et al. [83, 84] which uses a satisfiability modulo theory (SMT) engine (Hysat [71]) to search for bounds. SMT is an extension of Boolean Satisfiability (SAT) to other domains, including the set of real numbers. A SAT solver searches for an assignment of Boolean literals to prove a set of clauses holds true, or a proof of unsatisfiability. For example, the set of clauses $\{\{a, \bar{b}\}, \{\bar{a}, \bar{c}\}, \{b, c\}\}$ is satisfied by choosing $a, b, \bar{c}$. SMT extends this by searching for a proof or certificate of unsatisfiability that a set of clauses over real variables holds true. For example, for a set of clauses $\{a \in [0; 5], b \in [5; 10], a + b > 12\}$ is satisfiable, a certificate of this is the assignment $a = 5, b = 10$. The SMT solver works by propagating the initial intervals forward and backward, then refines these intervals by splitting until a proof of satisfiability or unsatisfiability is found [71]. The choice of where to split and how to perform a split constitute the main complexity of the solver. The

work by Kinsman et al. refines the bounds given by interval or affine arithmetic by searching for a set of inputs breaking the bounds using a SMT solver. This bound is iteratively refined depending upon the results of this test, using a binary search method. The main problem with this approach is that because the SMT solver internally refines bounds by splitting intervals to improve the bounds, it suffers from the scalability issues outlined for interval splitting. As a result of the slow run-time, the authors have attempted to improve bounds using vector approximations [82], and by adding additional constraints informing the solver to ignore certain regions [85], but both sacrifice the tightness of bounds.

The final set of tools based upon applying these techniques to improve bounds with interval analysis that are of interest are those which create output that can be verified by formal proof checkers. This is of value because the interval arithmetic operations will typically be performed in finite precision, and as such cannot formally guarantee a bound, whereas verifying a proof with a formal tool guarantees correctness. Most notably, this involves the work by Daumas et al. [40, 41, 43] where the initial tool created proofs which could be verified using formal tool PVS [120], the more recent tool known as Gappa creates proofs verified by Coq [144]. In the Gappa tool, a user specifies a check for a bound using a logical proposition, for example $\{x \in [0; 1] \rightarrow 4x - x^2 \in ?\}$, where ‘?’ asks Gappa to compute the best possible bound. This format allows a user to add extra detail via extra propositions if any is known, for example the exact bounds for an intermediate index variable in a constrained loop may be known exactly. Alternatively, it allows a user to ask Gappa if specified output bounds can be satisfied, for example $\{x \in [0; 1] \rightarrow 4x - x^2 \in [0; 4]\}$. It then proceeds using a set of in-built and user-defined ‘hints’ to prove a proposition. The in-built hints will typically be based on ideas as defined above, the user-defined hints will typically be limited to someone more familiar with interval analysis techniques to define specific re-formulations to help Gappa. Finally, if it succeeds, it generates a formal proof which includes all the hints it used, if it fails then the proposition may still hold, but Gappa is just unable to prove it, this is likely to be a result of the dependency problem and the use of interval arithmetic. An alternative version which is based on affine arithmetic, discussed below, aims to obtain superior results [94].

Taylor forms

Alternatively, more recently a set of methods that can be loosely grouped together under the name of Taylor forms, analysed in detail [115], have been developed which use a polynomial representation of the error terms, with the intuition behind this method being that this allows cancellation of dependencies; in the case of the basic example of finding the bounds of $x - x$, where $x \in [-2; 2]$, whereas interval arithmetic will return $[-4; 4]$, Taylor forms will return the true bounds of $[0; 0]$. Unfortunately, a polynomial with second order terms or higher contains dependencies within the polynomial, and finding optimal bounds for a multivariate polynomial has been shown to be NP-hard [38].

The most well-known of these Taylor forms, affine arithmetic (AA) [45], avoids this problem by restricting polynomials to first order, ensuring the polynomial contains no dependencies, meaning applying interval arithmetic to the final polynomial can find the ideal bounds. It works by representing every variable in an affine form given by (2.12) consisting of a known central value (c_0), coefficients of known value (c_i) and noise symbols (ϵ_i) that are only known to lie in the interval $[-1; 1]$ as shown in (2.12). The reason for choosing a central value is that it has been shown to have better error properties [130]: in the case of the earlier example (2.4), if we were to first re-write x as a centered variable, interval arithmetic would return the tighter bounds, as shown in (2.13). There are many other similar re-writing rules such as Horner form, power basis or Bernstein basis, which can be used to improve bounds [104]. The difference between using a centered form in affine arithmetic and creating self-validating methods is that the former only changes the affine polynomial which represents the range of any intermediate value in an algorithm, whereas the latter modifies the algorithm itself.

$$x = c_0 + c_1\epsilon_1 + c_2\epsilon_2 + \dots + c_n\epsilon_n. \quad (2.12)$$

$$\begin{aligned}
 x \in [0; 1] &\rightarrow x = 1/2 + x_1, x_1 \in [-1/2; 1/2]; \\
 4x &= 2 + 4x_1, x_1 \in [-1/2; 1/2]; \\
 x^2 &= 1/4 + x_1 + x_1^2; \\
 4x - x^2 &= 2 + 4x_1 - (1/4 + x_1 + x_1^2) = 7/4 + 3x_1 - x_1^2; \\
 &= 7/4 + [-6/4; 6/4] - [0; 1/4] = 7/4 + [-7/4; 6/4] = [0; 13/4];
 \end{aligned} \tag{2.13}$$

Affine arithmetic then performs all operations on these coefficients, ensuring the result is also in affine form. This method has been shown to be superior to other first order methods such as Hansen's generalised interval arithmetic [68], which is similar to affine arithmetic but replaces the coefficients (c_i) with intervals [45]. However, the problem with first order methods is that many functions, including general multiplication, are not affine, meaning approximations must be made on any higher order terms to ensure the polynomial only contains first order terms and still bounds the potential range. Methods to perform these approximations can trade the size of the error for computational complexity [158], but in all cases, any difference between the true range of the higher order terms and their approximation will result in wider bounds, and additionally, the dependency information between the higher order and lower order terms is lost. Furthermore, the error of these approximations is represented by adding a new noise symbol and this symbol can affect the scalability, as will be seen in Chapter 5.

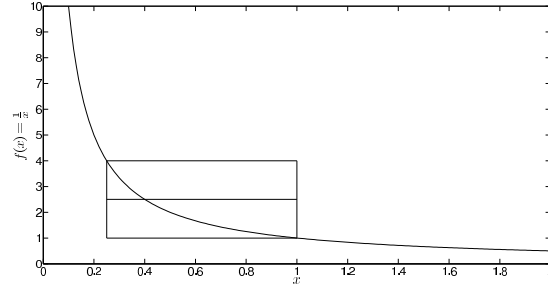
To minimise both these effects, the slightly more general method by Berz et al., named Taylor methods with Interval Remainder bounds (TwIR) [8, 103] represents range using the form (T_ρ, I_ρ) , where T_ρ is a polynomial consisting of all the terms that are less than or equal to an order (ρ) chosen by a user, and I_ρ is an interval which bounds the remaining higher order terms. Operations on functions in this form (T_ρ, I_ρ) are initially performed symbolically, before using interval arithmetic to evaluate any of the resultant terms involving the interval remainders and using an appropriate method to bound the new terms that are of degree greater than ρ , such as the Lagrange remainder [1]. Using a single interval I_ρ reduces the number of variables in the polynomial, whilst restricting the polynomial to higher orders ensures the errors arising from any approximations will be smaller, because all variables will be bounded less than 1. Furthermore, the choice of maximum order gives a user some form of control over the trade-

off between execution time and quality of bounds. However, representing the error with a single interval means any operations involving I_ρ suffer from the same dependency problem as interval arithmetic. In addition, finding the final bounds of the polynomial T_ρ still relies on interval arithmetic, and because T_ρ can contain higher order terms, dependencies will exist meaning it cannot find the tightest bounds. To illustrate this point, one can consider that interval arithmetic in its traditional form is equivalent to Taylor methods with Interval Remainder bounds where $\rho = 0$.

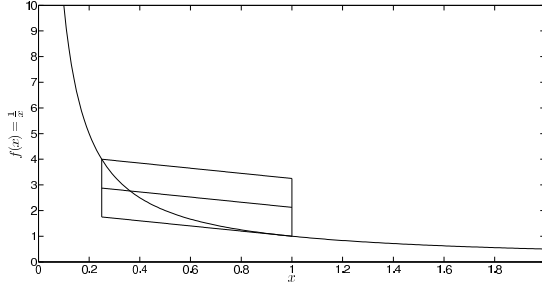
While it is possible to contrive simple examples where interval analysis returns better bounds than Taylor forms [124], in the absence of division Taylor forms can in general find tighter bounds than interval arithmetic by retaining first order dependencies. For example, Fang et al. [52,53] model fixed and floating point errors in the respective papers using a simple model, as in Section 2.2.1, and demonstrate that improved bounds by using affine arithmetic can be achieved in comparison with interval arithmetic, though they also illustrate that on more complex examples, affine arithmetic still fails to find tight bounds. Similarly Cong et al. [31] compare the range of various Taylor forms on simple polynomials to show they return tight bounds, though also point out that further improvements could be obtained by using automatic differentiation in conjunction with Taylor forms.

Polynomial approximation of inversion and other non-algebraic functions: Since Taylor forms operate on polynomials, in order for them to be applicable to more complex functions, it is necessary to approximate these functions with a polynomial. As an example, assume we wished to bound the function x/x , where $x \in [0.25; 1]$. This could be easily simplified to calculate the best bounds as $[1; 1]$, however, because x/x is a rational function, we would first have to create a polynomial approximation $\tilde{x}$ which bounds $1/x$, then bound the product $y \times \tilde{x}$. Figure 2.11 shows bounds for various methods to approximate the function $1/x$. The quality of the approximation is indicated by the total area which bounds the function. It is clear from these graphs that such approximations can induce a significant amount of error. Table 2.2 shows how these approximations would affect the final bounds of the function x/x .

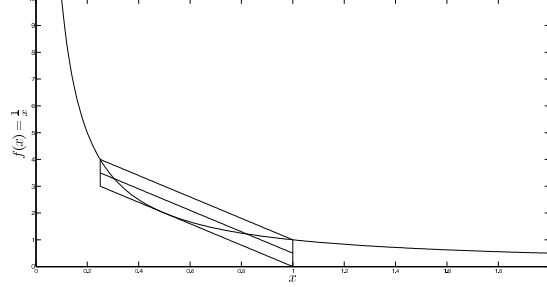
It is clear that the choice of approximation has a significant effect on the final bounds. There is a whole field of research in *approximation theory* in algorithms to create polynomial approximations of elementary functions, such as $1/x$, $\sqrt{x}$, $\sin x$, e^x , within this field, the most well known techniques



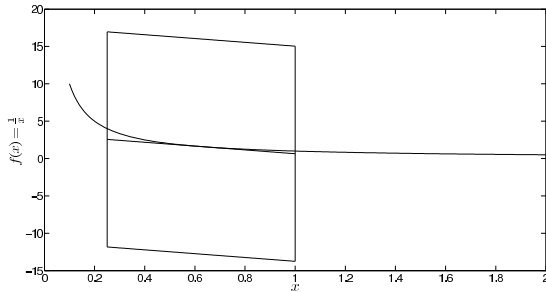
(a) Zero order (interval) approximation: $5/2 + 3/2\delta_1$



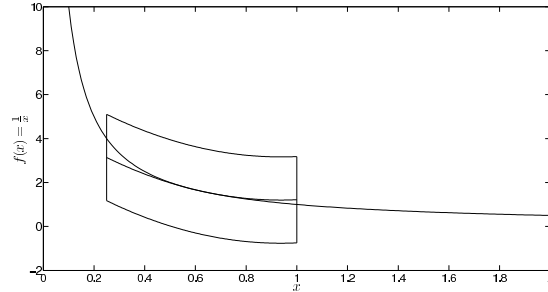
(b) Affine approximation using min range approximation: $5/2 - 3/8\delta_1 + 9/8\delta_2$



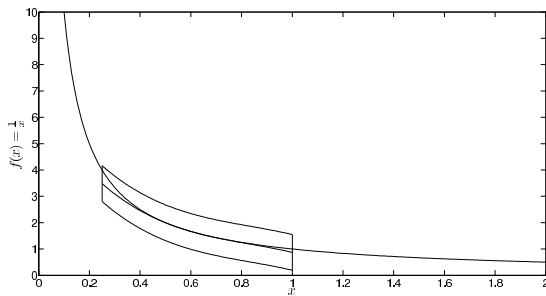
(c) Affine approximation using chebyshev approximation: $2 - 3/2\delta_1 + 1/2\delta_2$



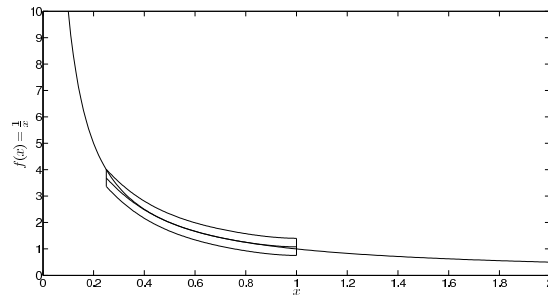
(d) Taylor approximation of order 1: $8/5 + 24/25\delta_1 + 72/5\delta_2$



(e) Taylor approximation of order 2: $8/5 + 24/25\delta_1 + 72/125\delta_1^2 + 108/55\delta_2$



(f) Taylor approximation of order 3: $8/5 + 24/25\delta_1 + 72/125\delta_1^2 + 216/625\delta_1^3 + 648/955\delta_2$



(g) Taylor approximation of order 4: $8/5 + 24/25\delta_1 + 72/125\delta_1^2 + 216/625\delta_1^3 + 648/3125\delta_1^4 + 283/872\delta_2$

Figure 2.11.: Approximations of $1/x$, $x \in [0.25; 1]$. The approximating function is written below the plot, where each variable $\delta_i \in [-1; 1]$.

Table 2.2.: Bounds of x/x .

Method	Lower Bound	Upper bound
Interval Arithmetic	0.25	4
Affine Arithmetic with min range	-0.4063	3.5313
Affine Arithmetic with Chebyshev approximations	0	2.5000
1st Order Taylor with Interval Remainder	-13.7600	15.7600
2nd Order Taylor with Interval Remainder	-1.1796	3.1796
3rd Order Taylor with Interval Remainder	0.1919	1.8081
4th Order Taylor with Interval Remainder	0.5977	1.4023

include Taylor approximations, Chebyshev approximations and the Remez algorithm [113]. Since this field of research is large, we only briefly outline these techniques here to provide a basic understanding. We comment that, in general, these methods trade quality of approximation with execution time.

Taylor approximations, as used for the Taylor with Interval Remainder method approximate a function using a Taylor series of a given order and bound higher order terms with the Lagrange remainder. While high order approximations can approximate a polynomial well, as seen in Figure 2.11, they take much longer to compute, especially for a multivariate polynomial. Furthermore, Taylor approximations are based around a single point meaning that for wide ranges, they may not approximate a bound effectively, as seen in the first order Taylor approximation of $1/x$ in Figure 2.11(d), which gives much wider bounds than the zero order interval approximation. Chebyshev approximations in contrast attempt to minimise the maximum error over the desired range of the function by choosing the coefficients c_n to minimise the difference between the desired function $f(x)$ and its Chebyshev polynomial of degree ρ , $\sum_{i=1}^{\rho} c_i T_i(x)$, where $T_i(x) = \cos(i \arccos(x))$. This often leads to better bounds, as can be seen by the first order Chebyshev approximation for affine arithmetic in Figure 2.11. The Remez algorithm is an iterative algorithm to find a minimax polynomial. It starts with a set of points, such as the roots of the Chebyshev approximation, and creates an approximating polynomial that ensures the difference between the approximating polynomial and the actual polynomial is equal in magnitude and alternating in sign at these points. It then chooses new points which reduce the magnitude of this difference and repeats this process. We note however that while there are a wide range of methods, the quality of approximation is heavily limited by the algorithm, for example affine arithmetic will only accept affine, or first order, approximations.

Arithmetic transform

The work by Radecka et al. [128] and Pang et al. [124, 125] attempt to improve bounds with the arithmetic transform. This performs range analysis using the so-called arithmetic transform, which converts the word-level representation into its binary digits. The initial work creates a branch and bound procedure which chooses values for individual bits for a given variable to avoid dependencies and compute the final range. The later work expands the ideas for use with Taylor series approximations, or to use it to reduce the errors in affine arithmetic approximations. However, by examining individual bits, the execution of this approach is large, of the same order of magnitude as the SMT approach, meaning it is not scalable.

2.2.3. Word-length optimisation

Having described the main techniques to perform range and precision analysis, we now briefly discuss work in word-length optimisation that has made use of these techniques to create custom hardware designs. The aim of an optimiser is to find either the minimum global bit-width or a set of bit-widths for each operation in an algorithm that satisfy a given final error specification.

There is a large amount of work which attempts to optimise word-length using simulation based approaches, for example, we have already mentioned various works which optimise the internal and global word-length in mixed precision solvers in Section 2.1.6. Further notable contributions in this field include the work by Sung et al. [87, 88, 142], which trade off system area with precision by iteratively modifying the word-lengths of signals whilst maintaining a given error over a set of benchmarks, measured by SNR. The methods to perform this modification include a heuristic search of word-lengths and an exhaustive search of word-lengths. The latter work groups signals to minimise the simulation time, and also attempts to modify word-lengths to be of the same size in the case of resource sharing. Cantin et al. [25, 26] built on this work and discuss various heuristics to perform individual word-length optimisation given the simulation data.

In terms of using analytical methods to optimise word-lengths in a system, early work includes the Bitwise Project by Stephenson et al. [137] which proposed a compiler based technique which creates data-flow graphs for an algorithm, and adjusts the bit-widths by forward and back propagation using

interval arithmetic. Back propagation is used when maximum bitwidths are known at mid points, such as the number of indices to an array, and this helps to further reduce bit-widths. This however is limited to finding the range, or choosing the position of the most significant bit. Nayak et al. [114] similarly use interval analysis to perform range analysis, but also perform error analysis using interval arithmetic to determine the number of least significant bits to satisfy a user specified bound on error at the output. The Précis project [27] adopts a similar approach of applying interval analysis, but it also compares this to a bit propagation approach. A simple example of bit propagation analysis is to compute the function $a = b + c$, with b and c 16 bits, a will need 17 bits. The bit propagation will always provide at least as large a range as the interval analysis, but the reason to perform it is to allow some slack if the initial estimated ranges were too small. The Précis tool estimates the impact of this slack on area to help choose which word-lengths are most important to constrain. Later work by the authors [28] add area and error models for adders and multipliers to this analysis so as to minimise the least significant bits and apply a simulated annealing based approach for automatic optimisation, where the user can steer optimising towards area versus error.

The problem with all of these works is that they are based on interval arithmetic, which as we have mentioned will overestimate range in the presence of dependencies. This has given rise to a set of mixed interval analysis and simulation based approaches. Early work by Bondalapati and Prasanna [18] mix bitwidth propagation plus run-time analysis to reduce number of bits. Keding et al. [81] and Willems et al. [152] develop the so-called FRIDGE compiler which improves on this by adding interval analysis to estimate the range, as well as bit propagation to choose the number of fractional bits. Similarly to the work by Stephenson et al., it then allows a user to ‘annotate’ intermediate variables, specifying limits on the range, maximum absolute and relative error or a specified bit sizes, to attempt to improve the bounds. The bounds are then checked against a user specified design criterion which may be maximum absolute and relative error or to satisfy a signal to noise ratio using simulation to ensure the user annotations are valid. Cmar et al. [30] adopt a similar approach of mixing simulation and interval analysis, the difference is that the latter approach advocates the idea that a user should be given information from both types of analysis, and be allowed to choose bit-widths accordingly. For example, if the simulated bounds are close to the interval analysis, it implies interval analysis is good and to use interval analysis, otherwise

it advocates creating hardware using the number of bits that were required according to simulation and to add error detection. In this work, the number of fractional bits are chosen partially by a user, who specifies some initial word-lengths, and partially by simulation which sets the number of fractional bits to ensure the error is less than the average standard deviation of error introduced by quantisation or simulated noise, calculated in floating point. Gaffar et al. [57] adopt a similar approach for floating point designs. Here they use interval analysis to estimate range, then simulation to search for excessively large ranges which are then backpropagated. This is repeated for all word-lengths in an algorithm until they have been reduced as much as possible. It also supports both global word-length and individual word-length optimisation to trade performance and silicon area. While global word-length optimisation is fairly straightforward - one can just reduce the global word-length until it violates the design criteria - to do individual word-length optimisation, it creates heuristics where the precision of every variable in the algorithm is reduced individually to see its impact on range, error and silicon area, and then chooses which word-length to reduce accordingly.

However, any combined simulation and interval arithmetic approach still loses robustness as it is not possible to exhaustively simulate over a set of inputs. One alternative is to use global optimisation strategies for interval analysis, for example, the work by Kinsman et al. [83, 84] minimises individual word-lengths using the SMT approach mentioned earlier, and later work uses area models to choose whether to implement operators using fixed or floating point or a combination [85]. However, as we have mentioned above, this has issues with run-time and scalability to large examples, as mentioned above. A second alternative to improve bounds is to use affine arithmetic. The Minibit tool [92, 93] is based upon a two-stage application of affine arithmetic, to first bound the range and secondly the precision. It performs it in this way to simplify the analysis, however, we note that a two stage approach will not be sufficient to bound the range for algorithms which include division as finite precision errors could have a significant impact on range as a divisor approaches zero, as we will see in Chapter 4. To minimise the total area of the circuit, it then adds an area model for each operator and performs a simulated annealing based upon this model and the affine arithmetic analysis. A variant upon this work is suggested in Minibit+ [118] which first suggests that, due to their respective limitations, a comparison of interval and affine arithmetic results should be used as this will sometimes return more favorable results.

Secondly, instead of simulated annealing based refinement, it associates area costs to every component and reduces bitwidths according to these costs, because this is much faster, albeit potentially creating worse quality hardware when optimising individual word-lengths.

Finally, the work by Constantinides et al. [33,34] restricts the problem to linear time invariant systems, which are systems that consist only of additions, subtractions and constant coefficient multiplications. In this problem domain, it is possible to analytically calculate the optimum number of bits using LTI theory. It then chooses individual word-lengths to minimise the area and satisfy a design criterion, such as signal to noise ratio, using a heuristic in the former paper and integer linear programming in the latter work. Unfortunately, even within the field of digital signal processing, there are many algorithms that cannot be classified as linear time invariant. As a result, further work attempted to use the same approaches to optimise word lengths and extended it to nonlinear systems by attempting to linearise the system around a point, but this relied on simulation around Taylor models of local perturbations at the point [32] meaning that it suffers from the same issues as any simulation based proof. However, we make a brief comment that this work also illustrated that it is possible to also obtain power savings through word-length optimisation.

2.2.4. Summary

In this section, we introduced the main number systems used in digital design, binary fixed point and binary floating point, as well as discussing the errors that can arise as a result of using these discrete number systems and simple models that can be used to bound round-off error. We then discussed methods that can be used to calculate bounds on the range or relative error of any variable within an algorithm, as a result of input ranges and an accumulation of these roundoff errors, highlighting the advantages and disadvantages of these approaches. Finally, we illustrated how various tools have made use of a combination of these methods to calculate bounds, potentially alongside some model of the area of the hardware components, to optimise the silicon area usage and power of hardware accelerators.

In terms of the word-length optimisation tools, they could be loosely grouped into a small number of categories: based on simulation, based on interval arithmetic, based on interval arithmetic and simulation, based on affine arithmetic and targeted towards LTI systems. To summarise the issues with

these approaches: anything based on simulation cannot produce a robust hardware design because corner cases could be missed; interval arithmetic results in excessive hardware as the bounds are not tight to the point that tools were created that combine interval arithmetic with simulation to attempt to detect wide bounds, but these suffer from missing corner cases as well; affine arithmetic can improve bounds to create better hardware, but the tools were still limited to DSP systems that did not contain division; tools could optimise word-length in LTI systems, but very few algorithms in scientific computing can be classified as LTI. Altogether, this implies that new methods are required to calculate bounds on the range or relative error of any variable within an algorithm, and as such, in Chapters 4 and 5 of this thesis we attempt to create new techniques to perform this function.

2.3. Background summary

This chapter has provided an introduction to the current field of hardware acceleration. This has included a broad discussion on general techniques used to accelerate algorithms with hardware, followed by a special focus on the existing tools and techniques available to automatically optimise the numerical precision used throughout a hardware design. In the initial discussion, we illustrated that the main factors exploited in a hardware accelerator were parallelism, pipelining, memory use and precision. We demonstrated that in the context of the conjugate gradient algorithm and its core operation, matrix-vector multiplication, that while pipelining can be used to ensure that an operator are used efficiently, the main the trade-offs in terms of performance are between parallelism and memory use, with various publications spanning this trade-off. In chapter 3, we expand on these ideas, demonstrating how to choose the number of problems required to exploit pipelining as much as possible to obtain a very high sustained performance and also show how to traverse this space of parallelism and memory use as well as adding some specific improvements. In this chapter, we have also shown that there is little work exploiting the benefits of precision, excluding those based on simulation, despite the fact that it can affect both of these factors.

In this chapter, we subsequently described how the errors that can arise from the use of finite precision arithmetic, and then discussed the analytical tools to calculate bounds on the range or relative error of a

variable in an algorithm. This section described the limitations of the existing methods from computing tight bounds in a tractable time. We did however discuss a range of tools that attempt to perform such optimisations automatically; these typically employ some heuristic or possibly integer linear programming to optimise the choice of precision depending on the results from one, or possibly a combination, of the analytical or simulation based bounding procedures. While they demonstrate performance benefits, due to the limitations of the bounding procedures, they have largely been restricted to small problems, or restricted problem domains, such as linear time invariant systems. Presumably this is the reason investigation into exploiting the benefits from tuning the precision in the hardware accelerators for sparse matrix-vector multiplication was largely limited to simulation. This provides an inspiration behind the work we describe in Chapters 4 and 5 where we attempt to create new methods that can compute tighter bounds in a scalable execution time for use in such an optimisation procedure to tune the precision in larger algorithms.

Chapter 3

Accelerating iterative algorithms using hardware

In this chapter, we present a case study of the use of hardware to accelerate algorithms. We have chosen to use FPGAs for this study because of their flexibility in comparison to general purpose processors and GPUs and ease of programming in comparison to ASICs in order to help enable us to examine various approaches to achieve hardware acceleration. The application we have targeted in this case study is the solution of a system of linear equations $Ax = b$ because this forms the basis of many problems in engineering and science, and as such there is a large interest in accelerating this application, both in software, resulting in a plethora of methods to calculate the solution to this problem [62], which we will briefly summarise in Section 3.1, and in hardware, leading to an increasing number of publications in this field [23, 46, 76, 96, 97, 105, 111, 145, 151]. In this chapter, we focus on accelerating one of these algorithms, the MINRES algorithm, of which there has been no previous publications on its implementation on an FPGA. We initially discuss how to create a dedicated acceleration circuit for this algorithm using an FPGA, before we examine how to achieve greater performance by accelerating the core component of this algorithm, computing the dot-product, by taking advantage of the specific properties of individual problem formulations and through the use of convex optimisation techniques.

Throughout the majority of this chapter, we have chosen to restrict the discussion to the use of IEEE 754 single precision floating point so as to focus the discussion on the various techniques to achieve high performance on an FPGA. While this is in keeping with related research in this field, in the final section of this chapter, Section 3.4, we discuss the limitations of the restriction and how this motivates our desire to create a new method which will allow us to tune the precision used throughout a hardware

implementation.

The main original contributions of this chapter are therefore:

- a demonstration of the suitability of the MINRES algorithm for use on an FPGA and an analysis of the design decisions and trade-offs involved to accelerate this algorithm [11],
- a design for solving multiple dense systems of linear equations in a pipeline for orders up to 145 using the MINRES algorithm, with results demonstrating a *sustained* performance, taking into account I/O overhead, of up to 53 GFLOPs, which is superior to comparable published work and provides an order of magnitude improvement over the peak theoretical performance of a comparable CPU,
- hardware architectures for banded matrices and symmetric matrices that can significantly extend the scalability to large order matrices and achieve higher degrees of parallelism [12],
- an optimisation strategy to reduce the number of embedded RAMs and maximise the parallelism, according to problem specification,
- hardware architectures that can trade parallelism with FPGA resources to achieve greater scalability,
- a discussion how finite precision arithmetic can affect the convergence of MINRES, and a discussion how finite precision arithmetic can affect the performance of a hardware accelerator for MINRES.

3.1. Solving a system of linear equations

The solution of a system of linear equations of the form $Ax = b$ (where A is an $N \times N$ matrix, while x and b are $N \times 1$ vectors) is a fundamental problem arising in a wide range of engineering and scientific fields. These fields include MIMO systems in communication [9] and control systems [102], as well as many general scientific computing tasks [70]. Though finding the solution to these systems will often only be a sub-computation within an algorithm, for example within optimisation problems [116] or

finding the solution to partial differential equations [112], it is typically one of the most computationally intensive parts of the whole algorithm. As such, an efficient method to find the solution to a system of linear equations is highly desirable.

There exists a large amount of research into efficient algorithms to solve these problems [62]. This research falls into two main families of methods: direct methods and iterative methods [5, 133, 148]. Direct methods are the more traditional methods to solve these problems; they find the solution in one shot, typically via some form of computationally intensive matrix factorization. Some examples of direct methods are Gaussian Elimination, Cholesky Decomposition and QR factorisation [62]. These algorithms are traditionally more popular as they are simple to understand and can find the correct solution after a determinate amount of execution time. However, in the realm of scientific computing these methods are beginning to become unpopular as their computation time is relatively fixed - the number of computations grow as of $\Theta(n^3)$, meaning the time to find solutions can become substantial for large systems.

Iterative methods provide an alternative to direct methods to solve these systems, and work by refining a solution with each iteration. The obvious value of this iterative refinement is that when initialized with a good guess, these methods can quickly converge on the solution. This is the case in many scientific computing problems, especially in optimisation [116]. Traditional iterative methods are algorithms such as the Jacobi, Gauss-Seidel and successive overrelaxation method. These are so-called stationary methods as they perform the same operation at each iteration on the current iteration vectors. Whilst these are simple to understand and implement, they are generally quite slow to converge [5]. However, in the past 50 years, a new set of non-stationary iterative methods which have iteration dependent coefficients have arisen. For an arbitrary input guess for the solution vector x , these methods are generally competitive with direct methods in terms of convergence, but they also have the potential benefits of early termination for a good initial guess [5].

There are many variants of these iterative methods, including conjugate gradients, MINRES, SYMMLQ, bi-conjugate gradients, QMR, Bi-CGSTAB, CGS, LSQR, and GMRES [62]. The choice of these methods will depend on matrix characteristics such that each method is most suitable for a specific class of problem. These methods trade computational complexity and storage for generality.

For example the conjugate gradient method is the simplest method and only needs to store the vector from the previous iteration, but it is only suitable for solving symmetric positive definite matrices as opposed to GMRES which must store vectors for all previous iterations but can be used to solve any system that is non-singular. The fact several methods exist for different matrices is an advantage of iterative methods; given information about the matrix it is possible to minimise computation time. The final advantage of these iterative methods over direct methods is that as well as being specialised to matrices, they can often be optimised for sparse matrices, a feature we exploit in Section 3.3.1. In contrast, despite developments on ‘direct sparse solution methods’, iterative methods are much more easily adapted to take advantage of sparsity [133].

3.1.1. The MINRES algorithm

The MINRES algorithm finds a (potentially approximate) solution x_k , to the system of equations $Ax = b$ (where A is an $N \times N$ matrix, x and b are $N \times 1$ vectors) by performing a minimisation of the residual $b - Ax_k$ in the two-norm over a Krylov subspace [62]. It will generally converge to a very accurate solution without the need to calculate the entire subspace and hence the subspace is built iteratively, using the Lanczos process [122]. Overall, the pseudo code is described in Figure 3.1.

The conjugate gradient method can also be interpreted as an algorithm that makes use of the Lanczos process and therefore there are some similarities between the two methods [62]. For cases where it is desirable to compare hardware implementations of these two methods, it is important to highlight the two major differences in terms of hardware costs, both of which are a result of working with the two-norm. Firstly normalisation is required, resulting in square root operations. Secondly, it results in a three-term recurrence as opposed to the two-term recurrence in the conjugate gradient algorithm; this increases storage requirements. Thus the MINRES algorithm trades an increase in circuit complexity for the ability to solve a wider class of problems.

3.1.2. Hardware implementations for solving a system of linear equations

The widespread use of the computation of a solution of a system of a linear equations in scientific computing has inspired substantial research into hardware acceleration of these algorithms. We have

```

% Initialisation
 $v_0 = 0$  ;  $v_1 = b - Ax_0$ 
 $\beta_1 = \|v_1\|_2$ 
 $\eta_{mr} = \beta_1$ 
 $\gamma_0 = 1$  ;  $\gamma_1 = 1$ 
 $\sigma_0 = 0$  ;  $\sigma_1 = 0$ 
 $w_0 = 0$  ;  $w_{-1} = 0$ 
 $i = 1$ 

while  $\eta_{mr} > \varepsilon_{mr}$ 
    // Calculate Lanczos Vectors
     $v_i = \frac{v_i}{\beta_i}$ 
     $\alpha_i = v_i^T A v_i$ 
     $v_{i+1} = A v_i - \alpha_i v_i - \beta_i v_{i-1}$ 
     $\beta_{i+1} = \|v_{i+1}\|_2$ 

    // Calculate QR Factors
     $\delta_i = \gamma_i \alpha_i - \gamma_{i-1} \sigma_i \beta_i$ 
     $\rho_1 = \sqrt{\delta^2 + \beta_{i+1}^2}$ 
     $\rho_2 = \sigma_i \alpha_i + \gamma_{i-1} \gamma_i \beta_i$ 
     $\rho_3 = \sigma_{i-1} \beta_i$ 

    // Calculate New Givens Rotations
     $\gamma_{i+1} = \frac{\delta_i}{\rho_1}$ 
     $\sigma_{i+1} = \frac{\beta_{i+1}}{\rho_1}$ 

    // Update Solution
     $w_i = \frac{v_i - \rho_3 w_{i-2} - \rho_2 w_{i-1}}{\rho_1}$ 
     $x_i = x_{i-1} + \gamma_{i+1} \eta w_i$ 
     $\eta = -\sigma_{i+1} \eta$ 
     $i = i + 1$ 
end

```

Figure 3.1.: MINRES Algorithm [55].

described many of these implementations in greater detail in Chapter 2, however in order to place the work described in this chapter in context with previous works, we once again summarise key features of our architecture against a selection of related work in Table 3.1.

This table highlights that through parallelisation and heavy pipelining of all floating point components it is possible to achieve a sustained performance of up to 53 GFLOPS on the Virtex5-330T, which compares favourably to other hardware implementations of floating point matrix inversion algorithms. Whilst some other implementations can process larger matrix orders, this is a result of the trade-offs with using off-chip RAM or exploiting sparsity. As a brief reminder of these issues, discussed in more detail in Chapter 2, sparse matrix solvers can handle much larger matrices as it is not necessary to hold or operate on any zeros in the matrix, but representing the matrix using a sparse format is likely to add significant overhead, and this is unnecessary for dense matrices. Using off-chip RAM to hold the intermediate results enables larger matrices to be held, but the I/O requirements to load onto the FPGA creates a bottle-neck (typically determined by the number of off-chip RAMs) upon performance and efficiency, typically leading to low speedups. In this implementation, it was of interest to answer questions regarding how to optimise the use of components and achieve maximum sustained performance through parallelism and pipelining problems. Therefore it was chosen to create a dense implementation and make use of the embedded memories on modern FPGA devices to buffer large matrices on chip to break the I/O bottleneck, at the cost of limiting the order of matrix to up to 145. Still, this order of matrix is considered relatively large for *dense* problems, is sufficient for many applications which depend upon matrix inversion [147], and could be used as a building block for solving larger systems. Moreover, it is 2 to 12 times larger than previous dense on-chip solvers. We also note that to date the majority of the research in this field has been restricted to the conjugate gradient algorithm [72], whereas we have created an implementation of the MINRES algorithm, which though closely related to the conjugate gradient algorithm is a more general solver, as discussed in more detail in Section 3.1. The motivation for choosing this specific algorithm is that it strikes a good balance between complexity and generality in that many scientific computing problems could be mapped to problems with symmetric matrices that are not necessarily positive definite [116]. Furthermore we note that because of the close relationship between iterative methods, the methods discussed in this chapter are likely to be applicable across different

hardware designs.

Table 3.1.: Comparison of floating point matrix inversion methods.

Method			Year	GFLOPS	vs. Software	Max Order	Sparsity	Off-chip RAM	Device	Matrix Characteristic
Direct	Gauss-Jordan	[46]	2006	N/A	$4 \times$	1700	Dense	Yes	Virtex II	Non-Singular
	LU	[145]	2006	2.6	$6 \times$	1000	Dense	Yes	Stratix II	Non-Singular
	QR	[151]	2008	35	N/A	12	Dense	No	Virtex5	Non-Singular
Iterative	CG	[111]	2008	N/A	$2.4 \times$	Large	Sparse	Yes	$2 \times$ Virtex II	SPD ¹
	CG	[97]	2008	35	$28 \times$	58	Dense	No	Virtex5	SPD
	CG	[96]	2008	N/A	$10 \times$	236	Banded	No	Virtex5	SPD
	CG	[96]	2008	32	N/A	92	Dense	No	Virtex5	SPD
	MINRES	this	2008	53	$10 \times$	150	Dense	No	Virtex5	Symmetric

¹ Symmetric positive definite (SPD)

3.2. Accelerating the MINRES algorithm using an FPGA

The optimal hardware implementation is dependent upon a number of factors - number of resources, latency (in terms of cycles per iteration), throughput and efficiency (in terms of the amount of time resources are in use). The design described aims to achieve a good balance in terms of optimising these factors. The following sections detail the main considerations and potential trade-offs between these factors and justifies any major decisions. To aid description for this section, a diagram of the circuit is shown in Figure 3.2.

Floating point units

There are currently many libraries of floating point operators for FPGAs, including Xilinx LogiCores [157], OpenCores [101], VHDL-2008 IEEE standardized floating point library [10], Southern California's cores [63], FloPoCo [42] and Northeastern University's cores [7, 150]. However, in this case study, we were more interested in the overall architecture design than maximising the performance by choosing the best floating point cores, and chose Xilinx Core Generator as this could easily be used with their development suite. However, it is important to mention that when creating the cores using these tools, we chose components with the maximum latency to obtain the maximum clock frequency, as this

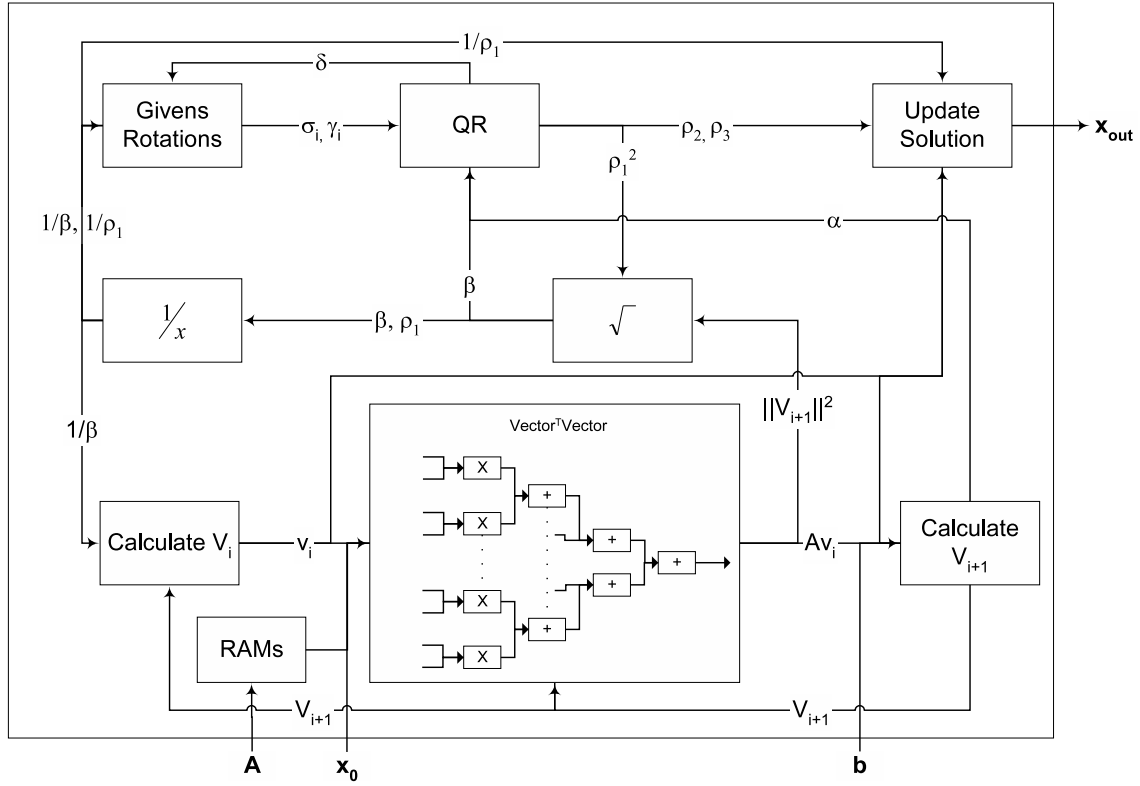


Figure 3.2.: Circuit data flow.

allowed the potential for a higher throughput, provided it was possible to keep the pipeline as full as possible.

We maximise the amount of time the pipeline is full by multiplexing P independent problems into the device. As a result, it was chosen to set all floating point components to work to their maximum latency, under the belief that implementation with a high throughput which could potentially operate on multiple problems simultaneously would generally be more useful than a small reduction in latency for a single problem. To this end, it was assumed that in all situations there would be sufficient problems available to the circuit to fill the pipeline. This assumption is valid in many problems requiring matrix inversion, for example for the control community [147]; we discuss the number of independent problems required to keep the pipeline below, where it is shown that it approaches 4 for large problems.

Parallelisation

It is clear from the pseudo code (Figure 3.1) that the calculation of the Lanczos vectors is independent of the operations to perform the QR decomposition, Givens Rotations and updating the solution. Therefore it is possible for all these parts of the circuit to work in parallel, and this reduces the overall latency to be that of the Lanczos iteration. We also briefly note that it is easily possible to perform the initialisation operations using the same circuit as for the Lanczos iteration because the only major processing is to calculate $\beta = \|b - Ax_0\|_2$, and the Lanczos iteration requires similar operations, meaning only a few additional multiplexers are required.

Theoretically, it is also possible to parallelise all matrix and vector operations, however, the limited number of resources on an FPGA mean that for large N it is not possible to parallelise every operation. As the operation of highest computational complexity is the *Matrix* $\times$ *Vector* multiplication within the Lanczos iteration, it is desirable to focus on the parallelisation of this operation. Though a dedicated component to perform this calculation would significantly reduce the latency of the circuit, the resource usage would scale heavily with N ($\Theta(N^2)$ in terms of multipliers and adders) and also the I/O requirements for such an implementation would quickly exceed the capabilities of the FPGA, making it highly unscalable. Instead it was chosen to overlap the *Matrix* $\times$ *Vector* operation in a pipelined fashion within a dedicated fully-parallel *Vector* ^{T} *Vector* or dot-product circuit. This involves a dedicated vector mul-

multiplier and an adder sum tree (as shown in Figure 3.2), which has a cost of N multipliers and $N - 1$ adders, but reduces the latency of the $Matrix \times Vector$ operation to be $\Theta(N)$ instead of $\Theta(N^2)$. In comparison, if one were to parallelise vector operations, it would only remove a constant latency at a cost of an increased use of N operators.

We also note that because we get a result for $Matrix \times Vector$ multiplication every N cycles using this circuit, using a single dedicated operator for each vector operation is sufficient to compute every vector operation in N cycles and this would keep the pipelines for these components fully in operation. It follows that a dedicated operator for each scalar operation will not fully be in operation and that it is possible to share these components. It also follows that if we were to perform α rows of the $Matrix \times Vector$ multiplication in parallel, such that it we would obtain the result of this operation in N/α cycles, then we would also need to increase the number of operators for every vector operation by α to match the latency of the vector operator with this circuit and avoid the need for pipeline stalls. For this reason, for the remainder of this section we have chosen to use a single fully-parallel dot-product circuit for this should scale to larger matrices as it consumes far less resources than a fully parallelised $Matrix \times Vector$ multiplication circuit, whilst still gaining a significant performance improvement through parallelisation. Furthermore, we have only chosen to take advantage of the potential to share scalar operators for the square root and division operations, because these consume a large number of resources, whilst using a dedicated component for additions and multiplications to minimise wiring and multiplexers between the floating point components ensuring the clock frequency is as close as possible to the maximum available for the floating point components. We perform vector division using a single scalar reciprocal followed by multiplication of this scalar result by the desired vector in order to share the resource to compute $1/\beta_i$ and $1/\rho_1$.

Having created this dedicated fully-parallel dot-product circuit, we note that it is possible to re-use this circuit when calculating the dot-products required when computing the results for $v_i^T A v_i$ and $\|v_{i+1}\|_2$ to potentially save resources. However, in order to perform this re-use, it is necessary to collate the entire vector in registers, ready to multiplex into the same circuit. For large N , the number of slices required to perform this may be larger than the number of slices needed to implement a small reduction circuit such as those described in Chapter 2. We therefore suggest to implement whichever method uses the lowest

resources.

Using this parallelism, the total number of floating point components is given in equation (3.1), with the potential addition of a reduction circuit for large N , and the overall latency of the circuit is given by equation (3.2), where P is the number of independent problems to be stored in the pipeline. Referring to (3.2), the factor $3N$ is a result of N cycles needed to perform the Matrix-Vector product in the pipeline as described above, as well as $2N$ cycles for the two series to parallel conversions (for v_i and v_{i+1}) which are input to this circuit (Figure 3.2); the factor $C_1 \lceil \log_2 N \rceil$ is a result of the summation tree in the $V^T V$ circuit (Figure 3.2). The values C_1 and C_2 are constants representing the latency of the other operations.

$$\text{Number of Floating Point Operators} = 2N + 26. \quad (3.1)$$

$$\text{Total Latency (cycles)} = 3N + C_1 \lceil \log_2 N \rceil + C_2. \quad (3.2)$$

Pipelining

As mentioned earlier, in order to maximise the performance of the circuit, the pipelines in the floating point components must continually be as full as possible, and this can be achieved by multiplexing P problems into the system. Due to the high resource usage of the fully-parallel dot-product circuit (N multipliers and $N - 1$ adders), in order to maintain a high performance, the minimum pipeline depth is chosen such that this component is always in operation. Furthermore, as discussed above, if the fully-parallel dot-product circuit is constantly in operation, all of the floating point operators for vector operations will also constantly be in operation.

Using the fully-parallel dot-product circuit described above, for P problems this circuit will be in operation for PN cycles, or $PN + 2P$ cycles if the fully-parallel dot-product circuit is re-used when computing $v_i^T A v_i$ and $\|v_{i+1}\|_2$ for one cycle per problem. Thus an effective way to determine the minimum pipeline depth is to match PN or $PN + 2P$ with the latency for one iteration (equation (3.2)), for this ensures the fully-parallel dot-product circuit will be operating on other problems until it is again

needed for the subsequent iteration of the first problem. Using this method, the pipeline depth is given by equation (3.3), and the depth of the minimum pipeline found by this method is shown in Figure 3.3. This graph shows that for small N , the number of problems needed to keep the pipeline busy is large, this is because the latency of computing the matrix-vector multiplication circuit is comparable to the subsequent operations, whereas when N gets larger, it is possible to perform the rest of the operations to calculate the result ready for the next iteration whilst the fully-parallel dot-product circuit is still in operation. The reason the pipeline does not monotonically decrease is that the latency of the fully-parallel dot-product circuit is increased as the depth of the vector-sum tree exceeds powers of 2. Finally, it should be clear from equation (3.3) that the depth of the pipeline tends to the value 4 as N tends to infinity, implying for large matrices only a small number of independent problems are required to keep the pipeline busy.

$$\text{Pipeline Depth } (P) = \left\lceil \frac{3N + c_1 \lceil \log_2 N \rceil + c_2}{N} \right\rceil. \quad (3.3)$$

We measure how effective this technique is in keeping floating point operators busy by estimating the efficiency of our circuit according to (3.4). This equation calculates the number of floating point operations that are performed when solving P problems and divides this by the potential number of floating point operations that can be performed if all the floating point operators in the circuit were always performing useful operations over this period. A graph illustrating the efficiency of the circuit using this minimum pipeline is shown in Figure 3.4. This demonstrates that in all situations a high efficiency (above 60%) is achieved and the efficiency increases with matrix order, tending to 100%. This growth in efficiency is due to the number of operators for the fully-parallel dot-product circuit increasing with N , whilst the number of operators working on scalars remains constant (equation (3.1)). This implies for large N , the number of operators is dominated by the fully-parallel dot-product circuit, and this circuit, along with the operators for vector operations, will always be in operation by design. Such a performance is highly unlikely to occur in any software implementation due to various delays such as cache misses. Indeed the efficiency of the order of 40 to 60 % is common in software, even for a highly optimized implementation, as shown in [48].

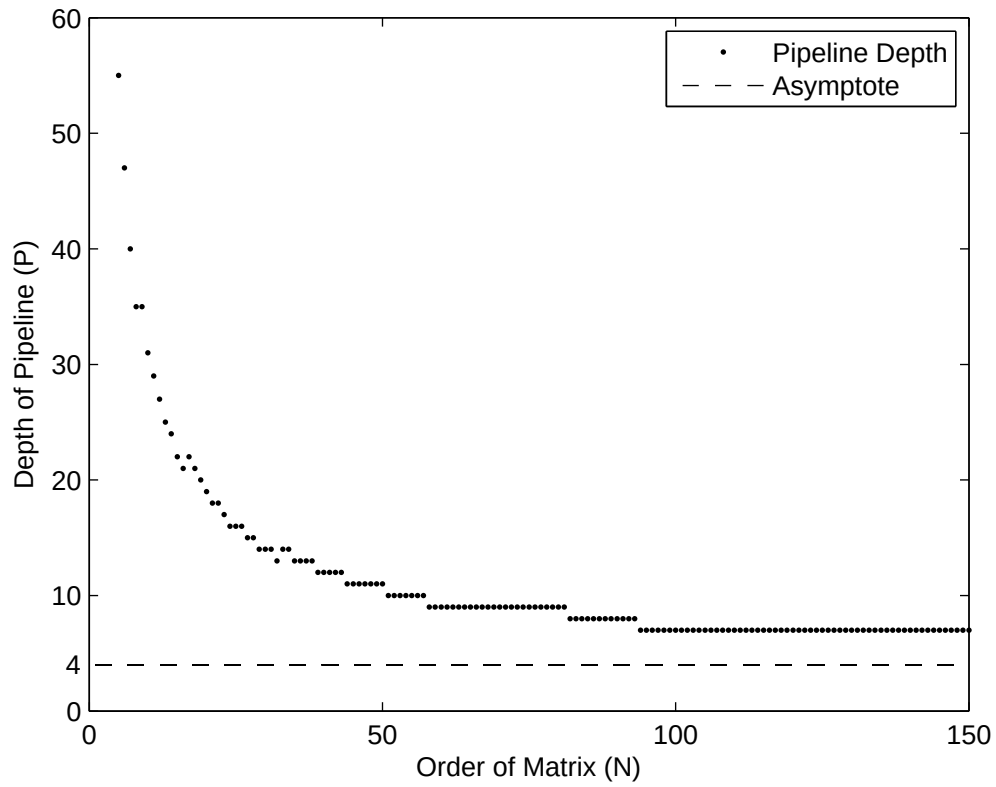


Figure 3.3.: Plot of pipeline depth for increasing matrix order. This is the minimum number of problems required for the $V^T V$ circuit to always be in full operation.

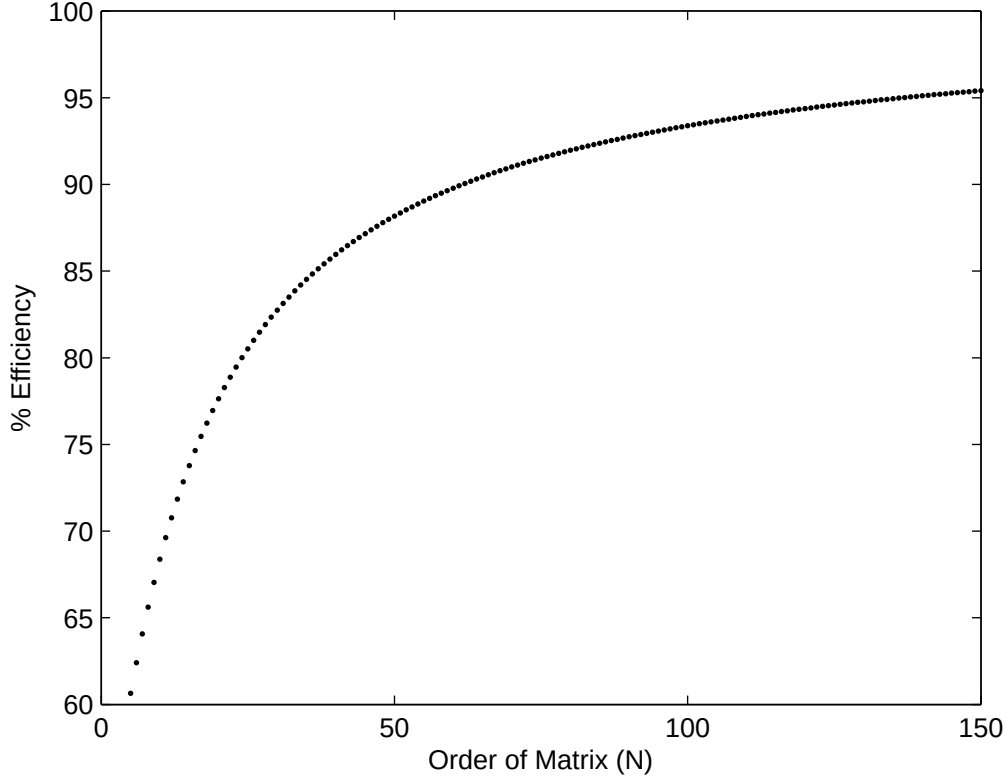


Figure 3.4.: Plot of percentage efficiency for increasing matrix order using the pipeline depth (3.3).

$$\text{Efficiency} = \frac{\text{Total Number of Floating Point Operations to find solution for } P \text{ Problems}}{\text{Total Number of Floating Point Operators} * \text{Total Latency}}. \quad (3.4)$$

I/O Considerations

The major consideration with regard to I/O is to ensure the fully-parallel dot-product circuit will continually have input data. As a result of using single precision floating point representation (requiring 32 bits) and the limited off chip I/O bandwidth in typical FPGA computing platforms, all elements of the A matrix cannot be loaded in parallel. Instead the A matrix is held in on-chip RAM (as shown in Figure 3.2), organised as a parallel bank of RAMs, each storing a column of the matrix for P problems.

The A matrix for a given problem is re-used during each iteration and hence the I/O requirement is

determined by the need to be able to load the set of A matrices into the FPGA for the next set of P problems within the time period to solve the first P problems. It is important to note that this method requires the RAMs to be twice as large as necessary for any single iteration (half of the RAM loads the next set of data whilst the other half is in use).

It can be shown that in infinite precision, the maximum number of iterations for a given problem to reach a solution is N [64], however, the method can often converge before this worst case. Denoting the number of iterations executed as I , then since the latency for one iteration, after matching for pipeline depth, is PN , the total time available to load the data is $I(PN + P)$. The total amount of data transferred is the A matrices, the two vectors b and x_0 , and the final output vector x_{out} for P problems; a total of $P(N^2 + 3N)$ elements. Thus the I/O requirement is given by equation (3.5a).

$$\text{I/O Req} = \frac{P(N^2 + 3N)}{I(PN + P)} \text{ words/cycle.} \quad (3.5a)$$

$$\approx \frac{N}{I} \text{ words/cycle.} \quad (3.5b)$$

$$= 1.1 \frac{N}{I} \text{ GBytes/s.} \quad (3.5c)$$

In order to consider this in terms of available I/O technology, this is also shown as Bytes/second (3.5c), using the clock frequency (Section 3.2.1). While I is data dependent in general, this I/O bandwidth is, in our experiments, well below that provided by typical FPGA computing platforms, such as PCI-express (8 GBytes/s).

3.2.1. Implementation results

Resource usage

The circuit was placed and routed, targeted to the Virtex5 LX 330T. Figure 3.5 shows the resource use in terms of DSP48Es, slices and BRAMs. The growth of slices and DSP48Es with matrix size is highly linear. This is to be expected, for the growth in floating point units is linear (equation (3.1)), and this design is dominated by floating point components.

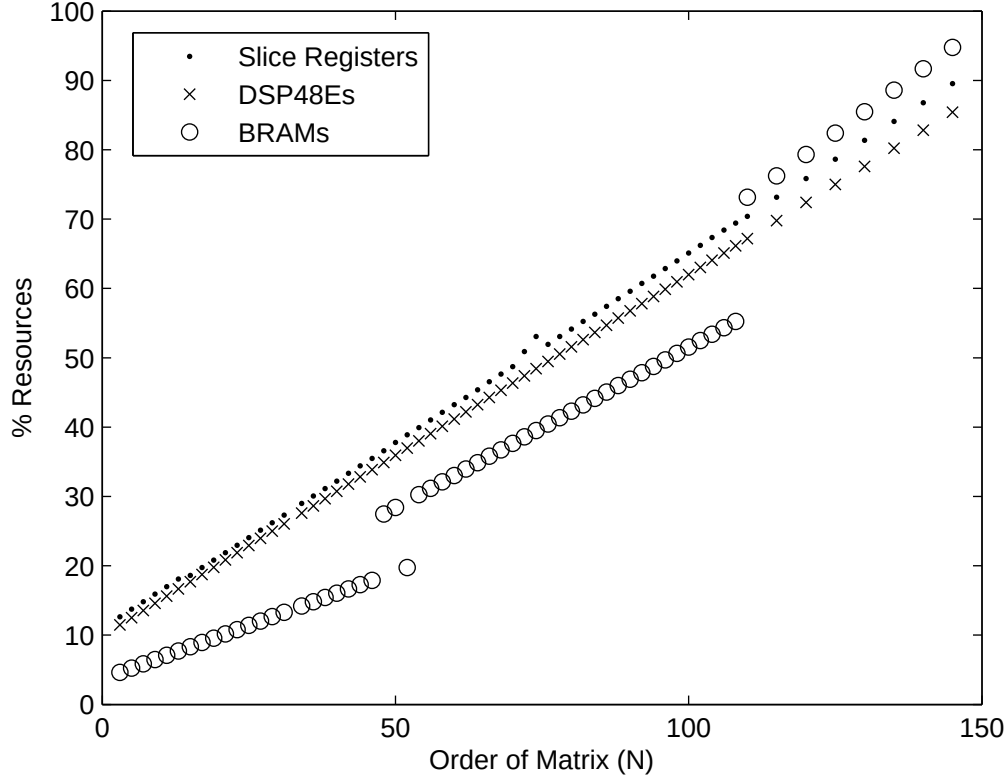


Figure 3.5.: Plot of percentage resource usage on a Virtex 5 LX 330T for increasing matrix order.

The BRAM usage, as seen in Figure 3.5, grows in a piece-wise linear fashion, with occasional jumps. This is a result of storing the N columns of the A matrix for P problems (which translates to N RAMs each storing PN elements) dominating the BRAM use. Thus linear growth is caused by the number of columns increasing with N , whilst the large jumps occur when PN exceeds the physical sizes of the BRAM. Together, this corresponds to a quadratic growth asymptotically, but for orders up to 145, this is not significant and is only seen as three jumps. The reason the BRAM usage is not monotonically increasing with matrix order is due to the decreasing pipeline depth (3.3) reducing the number of A matrices that must be stored.

Performance

The maximum clock frequency reported after place and route is approximately 250 MHz for small matrix orders ($N \leq 16$), after which the speed slowly degrades with N , approximately in a linear fashion, to about 175MHz for the largest matrix orders. This high performance is likely to be a result of the simple datapath, combined with the deep pipelining allowing a large degree of retiming freedom, whilst the degradation is likely to be a result of the increased size of the circuit requiring increased wiring. Given this frequency, the maximum matrix order of 145, the number of floating point operators given in equation (3.1) and the efficiency of 95% (Figure 3.4), it is possible for this circuit to achieve a sustained performance of around 53 GFLOPS.

As has been demonstrated in Section 3.2 this hardware implementation involves significant parallelism to reduce the latency of the iteration, and also works upon multiple problems in a pipelined fashion. In order to quantify this improvement, the performance of the hardware is compared to the peak theoretical performance of a software implementation [48]. The performance metric is MINRES iterations per second.

The software model is based upon the peak theoretical floating point performance of a Pentium IV running at 3.0 GHz (6 GFLOPS) [48], applied to the number of floating point operations given in equation (3.6) which is found by a simple operation count of the algorithm described in Figure 3.1. The hardware model assumes a pipeline depth given by equation (3.3) and the operational frequency given by the place and route results.

$$\text{\#Floating Point Operations} = 2N^2 + 15N + 14. \quad (3.6)$$

A comparison of these two models is shown in Figure 3.6. This shows that as a result of the parallelism reducing the latency, the performance is greater than a software implementation even for orders as low as $N = 7$. Due to the increased efficiency and parallelism, this performance improvement grows to almost an order of magnitude over the peak theoretical performance of a software implementation.

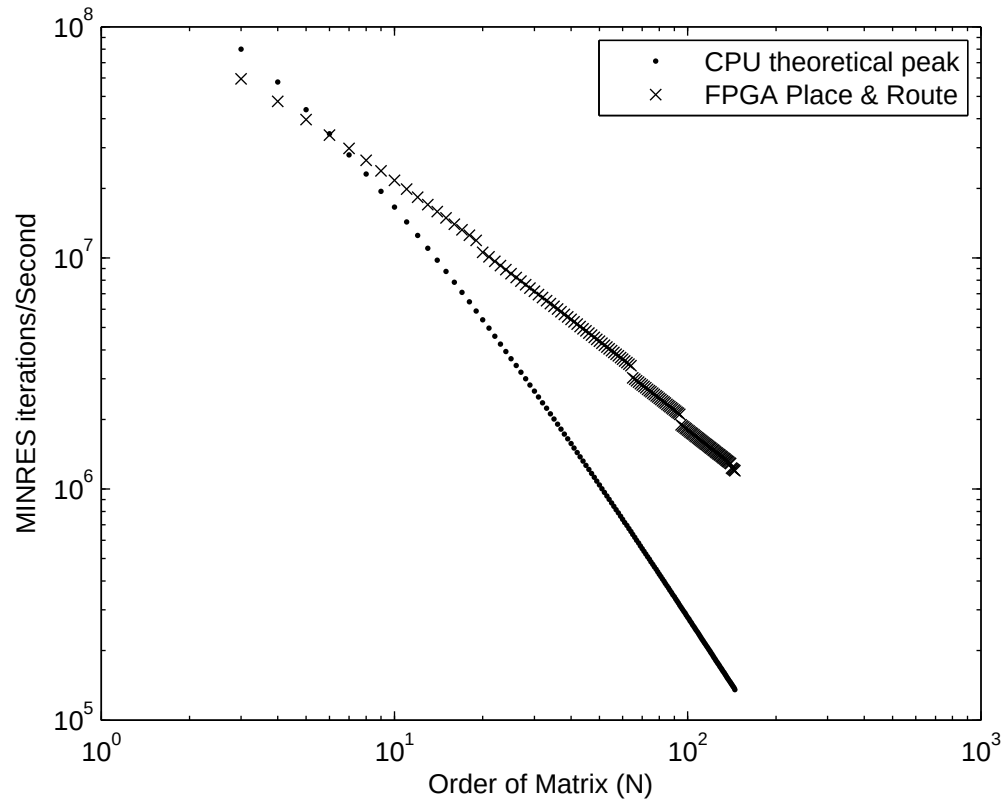


Figure 3.6.: Comparison of hardware and software performance.

3.3. Optimising memory bandwidth usage and performance for matrix-vector multiplication in iterative methods

We have shown that the MINRES algorithm can be used effectively as a means to solve a system of linear equations in hardware, and that through taking advantage of hardware parallelism, we could reduce the latency of the circuit by a factor of N and through pipelining it is possible to achieve an efficiency which will tend to 100%, with values of 95% achieved in practice, altogether obtaining a sustained performance of 53 GFLOPs, which is superior to previous work and predicts a performance improvement of nearly an order of magnitude compared to the peak theoretical software implementation. However, we also note that the high RAM use restricted the use of this accelerator to small matrices, whilst for small matrices, the performance is significantly below the peak.

Given many problems in scientific computing result in large matrices, it is of interest to determine the extent to which this performance can be maintained for such matrices, and it is similarly of interest to maximise the performance of the hardware accelerator for all matrix orders to obtain a high performance improvement for all matrix orders. As matrix-vector multiplication consumes the best part of the execution time of the algorithms, the main performance improvement was provided by the fully-parallel dot product circuit [74]. Since the major limitation to the accelerator was the requirement to buffer the input data on-chip so as to provide the desired bandwidth to feed the fully-parallel dot product circuit, in this section we examine methods to improve the memory use and maximise the parallelism of matrix-vector multiplication in general. To explore how this could be achieved, we adapted our hardware architectures for performing matrix-vector multiplication to take advantage of banded matrix structure, matrix symmetry, or both. Banded matrices are sparse matrices of a specific structure such that all of the non-zero values lie within a specified bandwidth of the diagonal, and these arise in many problems, for example when solving partial differential equations [135]. Symmetric matrices are square matrices equal to their own transpose, and these are of particular interest as both CG and MINRES algorithms will only converge to a solution provided the input matrix is symmetric [62].

We note that while there exist many methods to accelerate matrix-vector multiplication in hardware, as we discussed in more detail in Chapter 2, in this section we focus on improving the method we

Table 3.2.: Comparison of floating point matrix-vector multiplication in hardware.

Method		Year	Sustained GFLOPS	Device	On-chip RAM use method
SpMV	[50]	2006	1.5 single precision	Stratix S80	Stream from off chip
SpMV	[111, 161, 162]	2005-2007	2.16 single precision	Virtex2 Pro	Matrix off-chip using CSR, Vector on chip
SpMV	[47]	2005	12 double precision	16×Virtex2-6000	Sparse matrix on chip
MINRES ¹	this	2008	53 single precision	Virtex5 LX 330T	Dense matrix on chip, re-used over many iterations

¹ Majority of performance for MINRES circuit comes from matrix vector multiplication, see Section 3.2.

have used in this chapter, because it is through taking advantage of the ability to re-use of the same matrix over many iterations by storing the matrix on-chip, and subsequently making use of the greater memory bandwidth that the high performance of 53 GFLOPs is achieved. In contrast, methods for general sparse matrix-vector multiplication which use off-chip RAM are bandwidth limited and thus have limited performance. To re-illustrate this point, Table 3.2 compares several different methods to implement matrix-vector multiplication, and it is clear that by making increasing use of on-chip memory, the greatest performance is achieved.

3.3.1. Performing matrix-vector multiplication

This section describes simple modifications to the architecture shown in Figure 3.7 to solve matrices with specific structures using a high level of parallelism. We begin with a detailed description of banded matrices before describing the hardware architectures and RAM configurations to implement matrix-vector multiplication for this type of matrix, discussing in detail how this same approach can be used to handle both thin and wide bands. We then describe how this approach can easily be extended to handle symmetric matrices, reducing the RAM requirements, before discussing our procedure to optimise the use of RAM and LUT resources on an FPGA given this RAM requirement. Finally, we discuss our approach to trade parallelism for scalability for larger matrices.

Matrix-Vector Multiplication for Banded Matrices

Banded matrices are matrices where all the non-zero elements lie within some known bandsize M from the main diagonal, as shown in Figure 3.8(a). As the location of the non-zeros is known *a priori*, simple structures can be used to hold these values such as Compressed Diagonal Storage (CDS) [5], shown in Figure 3.8(b). This is preferable to other storage schemes such as compressed sparse row [5] because

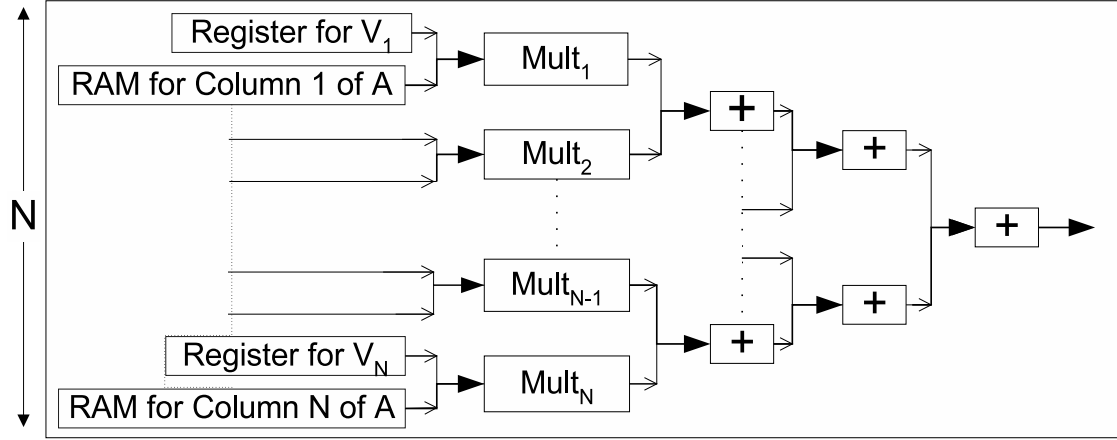


Figure 3.7.: Dot Product Circuit.

these schemes store indices of the locations of the non-zero elements, which is a waste of RAM use as this redundant information.

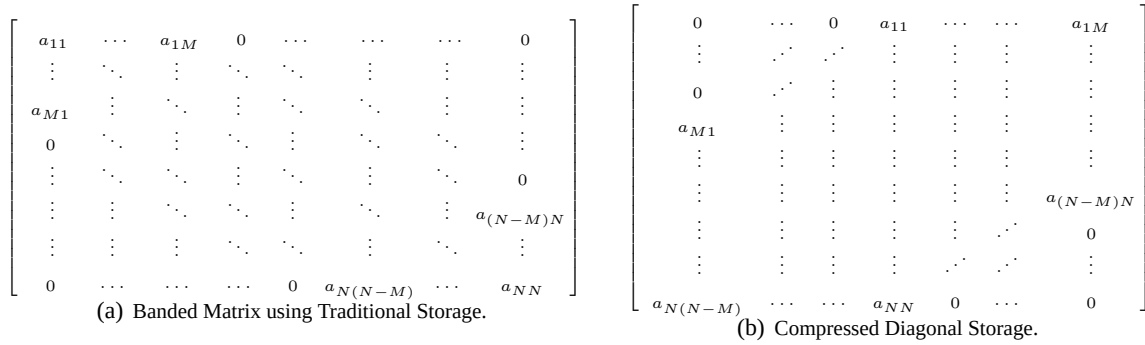


Figure 3.8.: Methods to store a banded matrix. Each column will be stored in a separate RAM, as shown in Figures 3.7 and 3.13.

Using CDS to store the matrix, all zeros that do not fall into the band are not stored, saving $(N - M)(N - M + 1)$ elements. However, as is clear from Figure 3.8(b), there are still some zeros in this storage. These zeros do not reflect any in the original matrix, rather they reflect the fact that at the band ends there are no elements and hence zeros are added instead. These total $M(M + 1)$ additional zeros. This implies that if $2M - 1 > N$, the amount of added zeros created from this redundancy could be greater than the number of zeros that are avoided by using this storage format. This section discusses these cases separately.

Thin-bands ($2M - 1 \leq N$): In comparison to the method for dense matrices, the first difference

is that instead of using N parallel multipliers, it is only necessary to perform parallel multiplications for the bandsize $(2M - 1)$, as the result of any other multiplications would be zero. The other slight complexity is that if the matrix is stored using CDS, the associated vector element for each RAM will change at each cycle. This is demonstrated in Figure 3.9 which shows the desired multiplications over time.

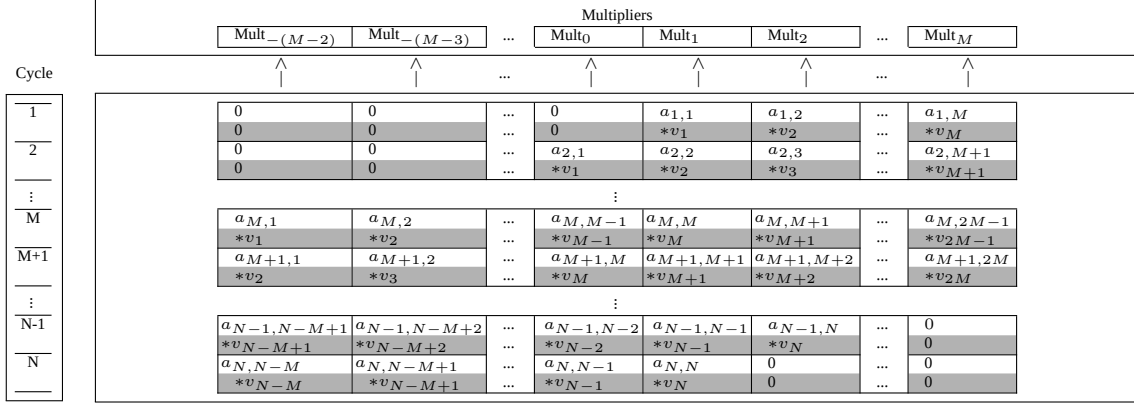


Figure 3.9.: Required multiplication over time. In this figure, the values in grey represent the required vector elements, whilst the values in white represent the required matrix elements from RAM. Any 0 value refers to a multiplication that need not be performed.

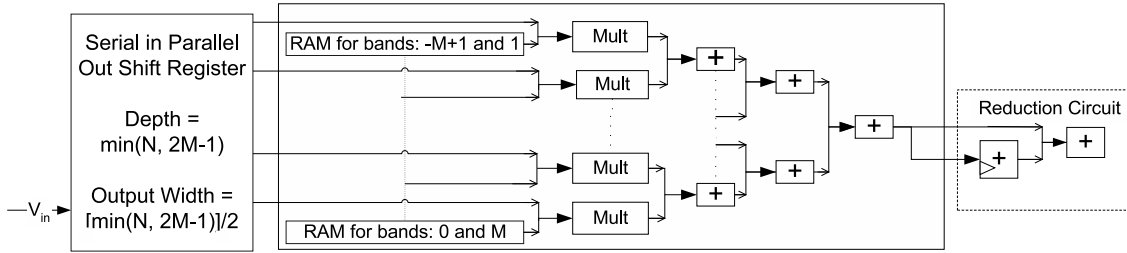


Figure 3.10.: Banded dot product circuit for thin bands.

However, from Figure 3.9, it should be clear that the required vector element for each multiplier is simply shifted once per clock cycle. This would require little additional hardware in comparison to Figure 3.7, which similarly to the work in [96] uses a vector of registers as the shift could be achieved using a serial-in-parallel-out shift-register, as shown in Figure 3.10. Furthermore, this shift register need only be of size $2M - 1$, as opposed to a vector of N registers.

Wide-bands ($2M - 1 > N$): There are two issues when using wide bands. The first is the excessive storage, as mentioned above, the other is that when using a banded matrix, the number of parallel multi-

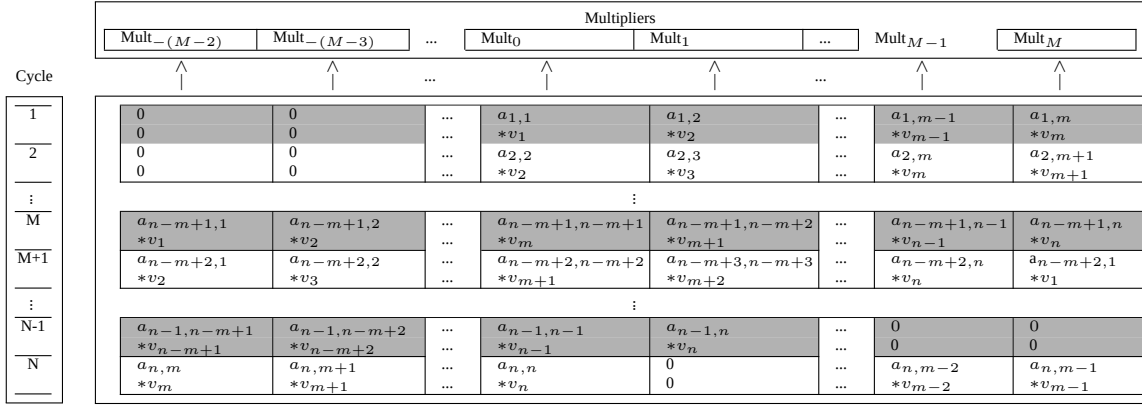


Figure 3.12.: Required multiplication over time for wide bands. In this figure, the values in grey represent the required vector elements, whilst the values in white represent the required matrix elements from RAM. Any 0 value refers to a multiplication that need not be performed.

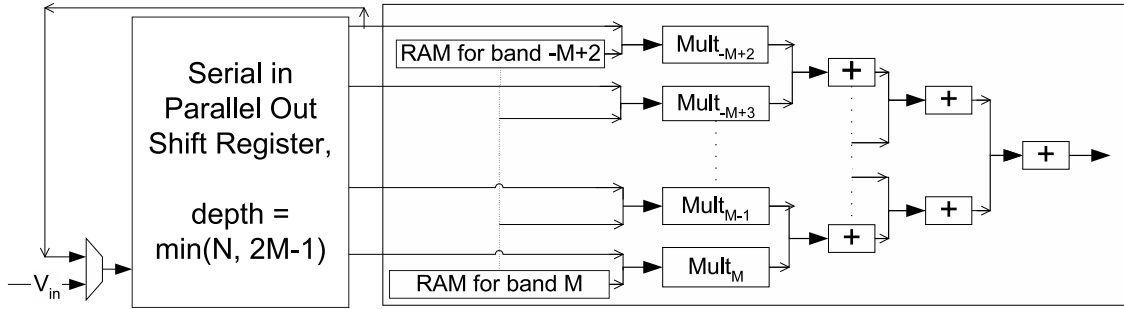


Figure 3.13.: Banded dot product circuits.

Matrix-vector multiplication for symmetric matrices

With symmetric matrices, it is only necessary to either store the lower or upper diagonal matrix. Interestingly, extending CDS (Figure 3.8(b)) to only store the symmetric portion is straightforward: all that needs to be done is to remove the columns that only hold the redundant data, *i.e.* all the columns to the left of that holding the diagonal. While this reduces the RAM requirements, ideally we would like to use the same architecture (Figure 3.13) because we believe it to have minimal control, since it consists of only a shift register, RAMs and a reduction tree, and this enables a high clock frequency, as well as ensuring that any architectures which improve these components of our circuit could be easily incorporated into our design. However, in order to achieve this, one must emulate the behaviour of the extra RAMs used for band storage. Interestingly, the organisation of the RAMs in CDS makes it quite simple to achieve this: as highlighted in Figure 3.14(a), the values to the left of the diagonal can be

seen as a delayed version of other columns, meaning these columns can be emulated using the required delays shown in Figure 3.14(b).

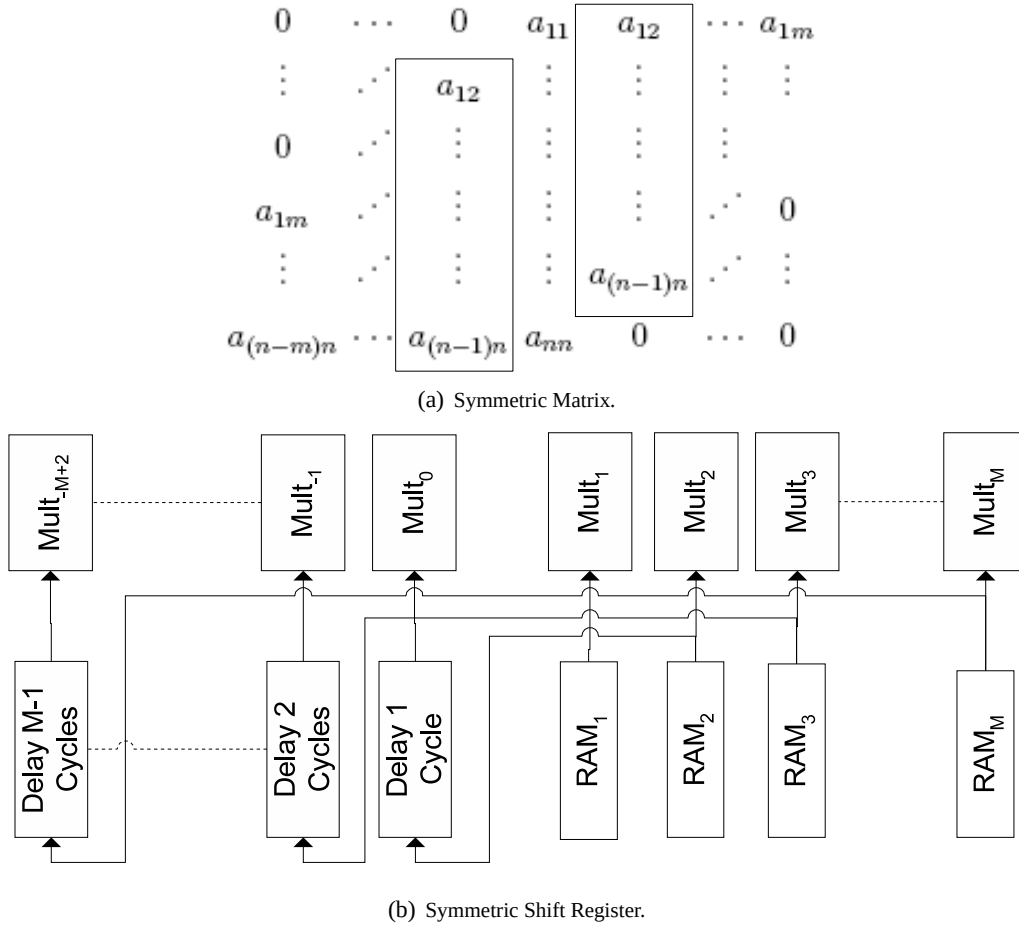


Figure 3.14.: Using Delays to Emulate Symmetry.

Implementing delays for symmetry on FPGAs: Using FPGAs, there are three potential methods to create this delay. The simple method would be to use FIFOs made up of either shift registers or RAMs. The problem with using this method is that if the delay is large, these FIFOs may also become large and this may use a lot of resources.

Alternatively, some FPGAs have embedded RAMs which can implement true dual-port memory. In this case, one port could access the current value, and the other port select the delayed value, meaning the delay could then be implemented simply by using a delayed counter which would require minimal additional circuitry. As it is not possible to describe an optimisation strategy for all FPGA architectures,

in this work we wish to demonstrate how one could make use of this additional functionality within the same optimisation framework, and we have chosen to use Virtex 5 family as a case study for this aspect; we believe that our framework could easily be modified when targeting a different architecture. However, there are more subtle issues when using Xilinx embedded RAMs. Xilinx BRAMs on a Virtex 5 are 36KBit, and can be configured in one of two ways: as 2-18KBit Block RAMs implementing simple dual-port RAM; or one single 36KBit true dual-port RAM [156]. This implies that by using the Block RAMs in true dual-port fashion, the amount of flexibility of the RAMs is reduced. Viewing this in another way, the likelihood of a large portion of an embedded RAM being empty is heavily increased, and this can reduce the number of RAMs available and impact the potential parallelism.

The final choice is that if the delay required for symmetry is greater than the size of RAM needed to store a column, then the same RAM can be used to also feed the multiplier for the symmetrical delay without requiring a second port, as only one of the two multipliers will require input data at any given time. Given these three options, determining the optimum use of resources is described in the following section.

3.3.2. Trading performance with slices

When implementing a matrix-vector multiplication circuit, ideally, the amount of parallelism should be limited by the resource constraints, which in turn will be dependent upon the chosen FPGA device and any other operations. In this section, we discuss two simple methods to trade the level of parallelism in order to make best use of available resources: increase the parallelism by performing parallel dot products or reduce the resource usage by performing dot products in stages.

Performing parallel dot products

In matrix-vector multiplication, it is possible to perform $\alpha \leq N$ parallel dot products for the multiple rows of the matrix. Whilst creating α parallel vectors of multipliers and adder reduction trees is a trivial task, the memory storage is not. The reason for this is that whilst it is necessary to have RAMs feeding each dot product circuit, it is undesirable to simply replicate the memory structure, for this would store a large amount of redundant data. Instead it is best to re-organise the data so that the RAMs only store

the data for the relevant dot product circuit. One simple method to do this would be to split a matrix vertically into α subsections and repeat the circuit. However, this is undesirable for two reasons: firstly, the delay circuitry for symmetry will then be repeated α times, which would waste hardware; secondly, to perform all operations in parallel, it would be necessary to have the entire vector available, which means that for a very large matrix with a thin band ($N \gg M$), there would be little gain compared to the approach specified in Figure 3.13 which begins performing dot products once the first band of the vector is available. Instead we wish to compute parallel dot products for α consecutive rows. We also note that this parallelism reduces the size of each RAM by α by spreading the RAM across several circuits. However, by doing this, we create dependencies between the various dot product circuits when implementing the symmetric delay; some example circuits which demonstrate how this is achieved are shown in Figures 3.15 and 3.16.

In order to understand how wiring and delay circuits are created, let us first label j as the circuit index, where $0 \leq j \leq \alpha - 1$, and i as the RAM column index, where $-M + 2 \leq i \leq M$. Now firstly, it should be clear from Figure 3.14(b) that because the RAM storing the diagonal ($i = 1$) is not re-used for symmetry, the first symmetric delay column ($i = 0$) will always come from the second RAM column ($i = 2$), with a delay of one cycle, and the second symmetric delay column ($i = -1$) will always come from the third RAM column ($i = 3$), with a delay of two cycles. We can extend this for all the symmetric delay columns, which are in the range $(-M + 2 \leq i \leq 0)$, to say they will delay data from the RAM column $2 - i$, with a delay of $1 - i$ cycles. Secondly, we must consider that the idea of the symmetric delay columns in Figure 3.14(b) is to store the data used in a previous multiplication, but when multiplying consecutive rows in different circuits in parallel, storing data from a previous multiplication corresponds to taking the data directly from the RAM of a previous circuit, and a delay of one cycle now corresponds to data from α multiplications back. This means that for a given circuit j , the first symmetric delay column ($i = 0$) can be taken directly from circuit $j - 1$ and the second symmetric delay column ($i = -1$) can be taken directly from circuit $j - 2$, whilst the symmetric delay column $i = -\alpha$ can be taken from circuit j with a delay of one cycle and the symmetric delay column $i = -\alpha - 1$ can be taken from circuit $j - 1$ with a delay of one cycle. In general, because the symmetric delay column i of any circuit j , as stated earlier, requires data from $1 - i$ previous multiplications, the

data for this column will come from circuit $j - (1 - i) \bmod \alpha$, with a delay of $\lceil (1 - i)/\alpha \rceil$ cycles.

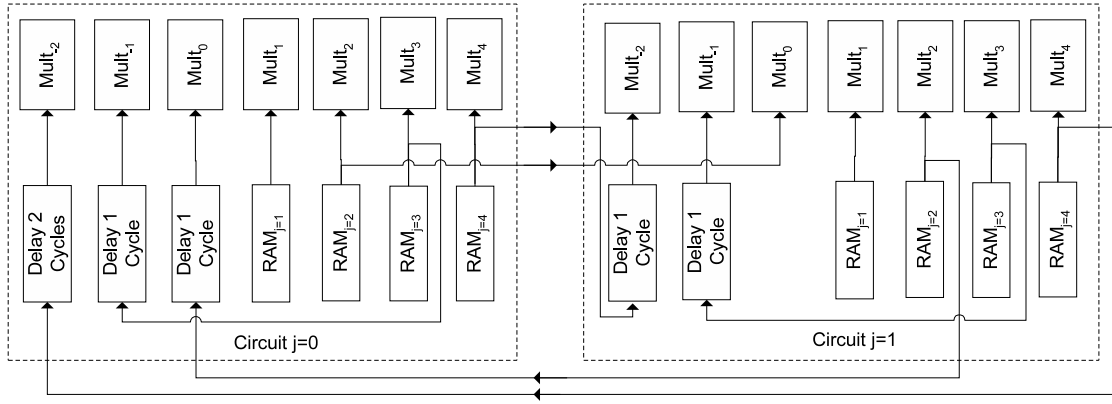


Figure 3.15.: $\alpha = 2$ Parallel Dot Product Circuits of Bandsize 4.

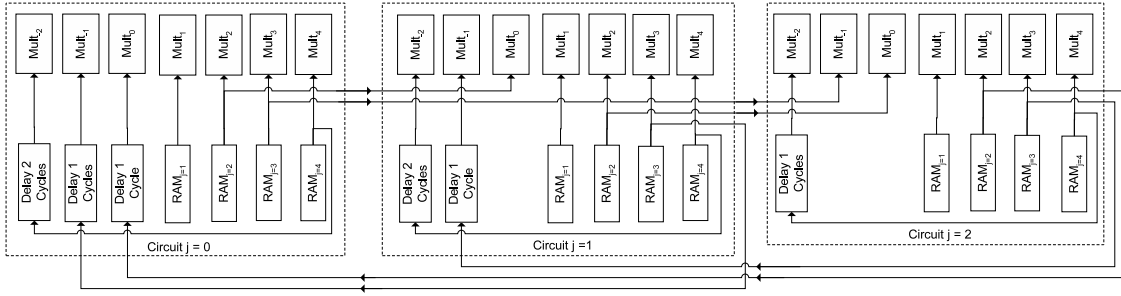


Figure 3.16.: $\alpha = 3$ Parallel Dot Product Circuits of Bandsize 4.

Performing dot products in stages

When creating a dot product circuit, the number of parallel multiplications grow according to $\min(N, 2M - 1)$ and the number of adders in the reduction tree grows according to $\min(N, 2M - 1) - 1$. This means that depending upon the available hardware, it may be necessary to restrict the parallelism to save slices, and we note that the significant reduction in memory use that this optimisation strategy provides implies that it would no longer be the memory available that limits the maximum matrix order of the matrix-vector multiplication, rather the slices and multipliers can easily become the limiting factor. We can reduce the amount of multipliers and adders used by performing partial multiplications of a row of size $\lceil \min(N, 2M - 1)/\beta \rceil$, where β is an integer, and use a reduction circuit, several of which are discussed in [161], to sum these partial multiplications. We note that by doing this, we reduce the number of re-

quired RAMs by β , but increase their size by β . The circuit required to achieve this is relatively simple, with the inputs for each multiplier controlled by a multiplexer. As an example, the circuit required for the case of $\beta = 2$, which would perform half the multiplication of the first half of the row during odd cycle and the second during even cycles, is shown in Figure 3.17. For different choices of β , only the reduction circuit would change, many of which are discussed in [161].

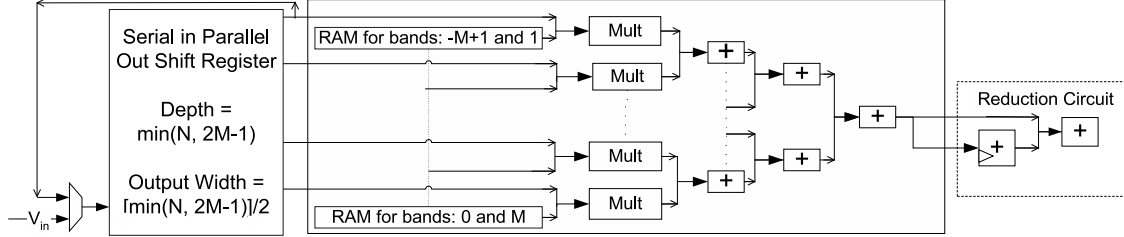


Figure 3.17.: Example circuit for performing dot products in stages, with $\beta = 2$.

Performing dot products in stages in parallel

Finally, it is worth noting that to obtain greater control over the trade off between parallelism and resources, it is possible to combine both of these methods. The reason this could be important can be demonstrated with a simple example. Suppose one was trying to perform dense matrix-vector multiplication for a matrix of order $N = 120$, but the number of slices limited the maximum number of floating point multipliers and adders each to 80. Obviously, one could not perform full dot product, as there is insufficient resources. Instead, it is possible to perform a partial dot-product by choosing $\beta = 2$, which would use 60 multipliers and 59 adders. This would work, but it fails to make the best use of resources, for alternatively we could perform $\alpha = 2$ dot-products in $\beta = 3$ stages in parallel, each partial dot-product circuit would use 40 multipliers and 39 adders, and an extra adder would be needed for the reduction circuit, using almost all the available resources.

3.3.3. Maximising matrix-vector performance

The amount of RAM required to store the matrix is dependent upon the the size of matrix N , the bandsize M , and the number of slices on the FPGA the user is willing to allow to be used in the place of embedded RAMs, whilst the amount of parallelism is also restricted by the number of LUTs and DSPs

(or embedded multipliers) on the FPGA the user is willing to allow to be used to create the circuit, as well as the number of required RAMs to feed the circuit. In our circuits, we have implemented single precision floating point operators using Xilinx Coregen [156], with each multiplier using a single DSP. Given all these constraints on the RAM requirements, LUTs and DSPs, we wish to answer the following questions:

- “For a given matrix of order N and bandsize M , and provided the FPGA device can provide a specified number of available BRAMs R , slices S and DSPs D : what is the maximum parallelism that the circuit could obtain?”
- “For a given matrix of known bandsize M , and provided the FPGA device can provide a specified number of available BRAMs R , slices S and DSPs D , then if there is a minimum requirement on the amount of parallelism: what is the largest possible matrix order that the circuit could solve?”
- “Once we have solved one of the above questions and worked out the maximum parallelism or maximum matrix order for the circuit: which configuration (making use of true dual port RAMs, using single port RAMs or using registers) should we use to store a column of the matrix or implement the delay for symmetry?”
- “Once we have solved all of the above questions: how many parallel dot product circuits should we create (what is α ?), and how many stages should we take to perform each dot product (what is β ?)?”

To answer all these questions automatically, this section proposes an integer linear programming (ILP) formulation which can be solved by many existing solvers, such as CPLEX [79]. The ILP is shown in Figure 3.18, this section discusses how the ILP is obtained.

Notation: As shown in Figure 3.8(b), the matrix columns in CDS contain trailing zeros. It is not necessary to store them, and hence the RAM requirement for each column changes. As a result, the variable i is used as an index for the M columns, with $i = 1$ being the column containing the diagonals. Similarly, as a result of the fact that, as mentioned in Section 3.3.2, the depth of each RAM will also vary depending upon our choice of the size of the partial dot product and whether it is shared across

Given an input matrix of order N and bandsize M for P problems, and user input constants X, Y, Z (defined in Section 3.3.3)

$$\begin{aligned}
 \mathbf{max:} \quad & Z\alpha - (1 - Z) \sum_{i=1}^{MY} (2\rho_i + \sigma_{1i} + \sigma_{2i}) && (\text{Maximise Performance}) \\
 \mathbf{subject\ to:} \quad & \\
 & \forall i \forall j, 2B\rho_{ij} + B\sigma_{1ij} + \tau_{1ij} + L \sum_{k=1}^j (\nu_j - 1) \geq Y\beta P(N-i+1)/X && (\text{Matrix Memory Constraints}) \\
 & \forall i \in \{1, \dots, N\} \forall j, 2B\rho_{ij} + B\sigma_{2ij} + \tau_{2ij} + L \sum_{k=1}^j (\nu_j - 1) \geq Y\beta(i-1)/X && (\text{Symmetric Delay Constraints}) \\
 & \sum_{i=1}^{MY} (2\rho_i + \sigma_{1i} + \sigma_{2i}) \leq R && (\text{Available RAM Constraints}) \\
 & K_1 \sum_{j=1}^{\alpha} (\tau_{1ij} + \tau_{2ij}) + \iota K_2 + \kappa K_3 \leq S && (\text{Available Slice Constraints}) \\
 & K_4 \iota + K_5 \kappa \leq D && (\text{Available DSP Constraints}) \\
 & \iota - \alpha \min(N, 2M - 1)/Y = -1 && (\text{Required Adder Constraint}) \\
 & \kappa - \alpha \min(N, 2M - 1)/Y = 0 && (\text{Required Multiplier Constraint}) \\
 & \forall j, \alpha + L\nu_j + L \sum_{k=1; k \neq j}^X (1 - \nu_j) \geq j, \quad \sum_{j=2}^X \nu_j = X - 1 && (\text{Parallelism Constraint 1}) \\
 & \forall j, \alpha + L\nu_j + L \sum_{k=1; k \neq j}^X (1 - \nu_j) \leq j && (\text{Parallelism Constraint 2}) \\
 & \forall j, \beta + L\nu_j + L \sum_{k=1; k \neq j}^X (1 - \nu_j) \geq X/j && (\text{Parallelism Constraint 3}) \\
 & \forall i \forall j, \rho_{ij}, \sigma_{1ij}, \sigma_{2ij}, \tau_{1ij}, \tau_{2ij} \in \mathbb{Z}, \quad \iota, \kappa \in \mathbb{Z}, \quad \alpha \in \mathbb{Z}, \\
 & \forall j, \nu_j \in \{0, 1\}, \quad i \in \{1, \dots, MY\}, \quad j \in \{1, \dots, X\}
 \end{aligned}$$

Variables	
ρ_{ij}	No. TDP RAMs used in column i of circuit j
σ_{1ij}	No. SDP RAMs used in column i of circuit j
σ_{2ij}	No. SDP RAMs used in symmetric column i of circuit j
τ_{1ij}	No. shift regs used in column i of circuit j
τ_{2ij}	No. shift regs used in column i of circuit j
α	Number of parallel circuits chosen
β	Size of required RAMs
ν_j	Indicator variable of whether a parallel circuits is used
ι	No. adders used
κ	No. multipliers used
Constants	
R	Max No. RAMs on Device
S	Max No. Slices on Device
D	Max No. DSPs on Device
B	Number of words stored per BRAM
K_1	Number of slices per shift register
K_2	Number of slices per adder
K_3	Number of slices per multiplier
K_4	Number of DSPs per multiplier
K_5	Number of DSPs per multiplier

Figure 3.18.: Maximising performance using ILP.

parallel circuits, the variable j as an index for each parallel circuit. To obtain control over the size of the ILP, we also add input constants X and Y for the maximum number of the parallel circuits and smallest partial dot products respectively, with the variables α and β are used to determine the chosen number of parallel circuits and the size of the required RAMs, whilst the binary variables ν_j are used conjunction with a large value L for ‘big-M formulation’ [153] which represents the decision over whether a parallel circuit is used.

There are three choices to store matrix elements and to implement the delays: true dual-port RAM, simple dual-port RAM or shift-registers, and as described in Section 3.3.1, true dual-port RAM can both store matrix elements and implement symmetric delay, whereas separate simple dual-port RAM or shift-registers are needed to implement this delay. To simplify the notation, for column i of circuit j , we use integer variables which represent the number of true dual-port RAMs, simple dual-port RAMs, simple dual-port RAMs used for delay for symmetry, shift-registers and shift-registers used for delay for symmetry, are denoted as $\rho_{ij}, \sigma_{1ij}, \sigma_{2ij}, \tau_{1ij}, \tau_{2ij}$ respectively. As the RAMs and shift registers are physical components, these must be integer variables, making this an ILP.

The remaining input constants are $R, S, D, B, K_1, K_2, K_3, K_4, K_5$ and integer variables are ι, κ . The constants R, S, D represent the maximum number of BRAMs, slices and DSPs on the target device, whilst the rest are used to translate these integer variables into the actual components: the maximum capacity of the BRAMs in terms of the number of words they can store is denoted as B , K_1 represents the number of slices to create a one cycle delay, K_2, K_3 represent the required number of slices to create an adder or multiplier and K_4, K_5 representing the required number of DSPs for these components. If it is possible to create several different adders and multipliers which trade use of DSPs and slices, extra variables could be added alongside ι and κ , accompanied with associated constants indicating their DSP and slice usage.

Objective function: The aim of the ILP is to maximise the performance, and minimise the RAM use. The first objective involves creating the largest number of parallel multiplications, given by α . For the second objective, the RAM use is a summation of the variables for the various RAMs whilst noting that as the true dual-port RAMs are twice the size of the simple dual-port RAMs, the cost for all ρ_{ij} variables is twice that of σ_{1ij} and σ_{2ij} . In order to convert both goals into maximisations, the latter term

is negated. Finally, as both objectives will compete for slices, multipliers and RAMs, the variable Z is added as an input variable to allow a user to favour one goal over the other.

Matrix memory and symmetric delay constraints: The three types of storage must satisfy the matrix memory and symmetric delay constraints for each column i and circuit j . The big-M formulation in this constraint ensures that if parallel circuit j is not used, there will also be no RAM constraint. The complexity in this approach is that, as mentioned in Section 3.3.1, the Xilinx true-dual port RAMs contribute to both the symmetric delay and matrix memory constraints, and thus this same variable appears in both inequalities.

The RAM requirement for each column of a matrix incrementally decreases, and hence the memory requirement could be given by $(N-i+1)$. However, in previous works [11,47,97] it has been highlighted that due to the deep pipelines in floating point operators on FPGAs, in order to maintain high sustained performance it was necessary to perform matrix-vector multiplication on many different problems in a pipeline, and each of these problems would have to be stored in RAM. This can easily be incorporated into the model by modifying the memory requirement for P problems to be $P(N-i+1)$. In contrast, as i increases, the symmetric delay requirement for each column increases incrementally, but for delays, it is no longer necessary to store multiple problems, and hence the symmetric delay requirement is generally given by $i-1$. However, as mentioned in Section 3.3.1, if $i > N$, then there is no need for a separate RAM to implement the extra delay, and hence the symmetric delay requirement is $i-1$ if $i \leq N$ and 0 if $i > N$.

Finally, one should note that by replacing symmetric delay constraints with extra memory constraints, it is possible to use this same ILP for banded matrices.

Available RAM/Slice/DSP constraints: These constraints translate the integer variables into resource constraints which ensure the implementation will fit on an FPGA. For the RAM constraint, the integer variables for TDP rams ρ_i have twice the weight as they consume twice the number of resources. For the slice and DSP constraints, the integer variables ι and κ for the number of adders and multipliers implemented are weighted by the estimates for the resource use for these components.

Required adder/multiplier constraints: These constraints ensure that there are sufficient multipliers and adders to implement the relevant level of parallelism, noting that the required number of adders

is one less than the number of multipliers.

Parallelism constraints: There is an inverse relationship between the size of the RAMs and the number of circuits, and this cannot be directly represented in an ILP. However, as we have restricted the maximum amount of parallelism and minimum dot product size, there is a distinct number of configurations and thus we can use big-M constraints to ensure that for any level of parallelism (α), the required RAM size (β) satisfies this inverse relationship. The big-M constraints are formulated such that if the binary variables $\nu_j = 1$, then the parallelism constraints are satisfied. However, because one ν_j must equal 0, then the number of parallel circuits α will be forced to the value j by parallelism constraints 1 and 2, while parallelism constraint 3 will ensure β is large enough to support the chosen level of parallelism.

3.3.4. Results

RAM use

The main benefit of this work is that it significantly reduces the RAM use. We demonstrate this by using our ILP to optimise the RAM use for four banded matrices of varying widths. Figure 3.19 consists of four graphs showing the percentage of embedded RAMs of a Virtex 5 LX330T that are required to hold these matrices using traditional dense storage, and by using band storage or symmetric band storage when optimised using our ILP. In these examples, the number of pipelined problems has been set to $P = 10$ so as to be comparable to previous works [11], and we set $Z = 0$ to focus on optimising RAM use. In most of the test cases, we have set $X = 1$ and $Y = 1$ so as to perform comparisons with the traditional method of storing a dense matrix. However, for cases where it is necessary to reduce the parallelism in order to satisfy the slice constraints of the FPGA, we allow $Y = 2$. These choices also ensure a short run-time for the ILP of a few seconds. Finally, in order to perform a fair comparison with the dense implementation, we allow the slices on the FPGA that are not used for the dot-product circuit to store matrix elements, a procedure which is performed automatically in our optimisation framework for the banded and symmetric banded cases.

The greater scalability of this approach is clearly shown for the thinnest bandsizes, when $M = 20$, which demonstrates that a large amount resources can be saved in comparison to the basic method

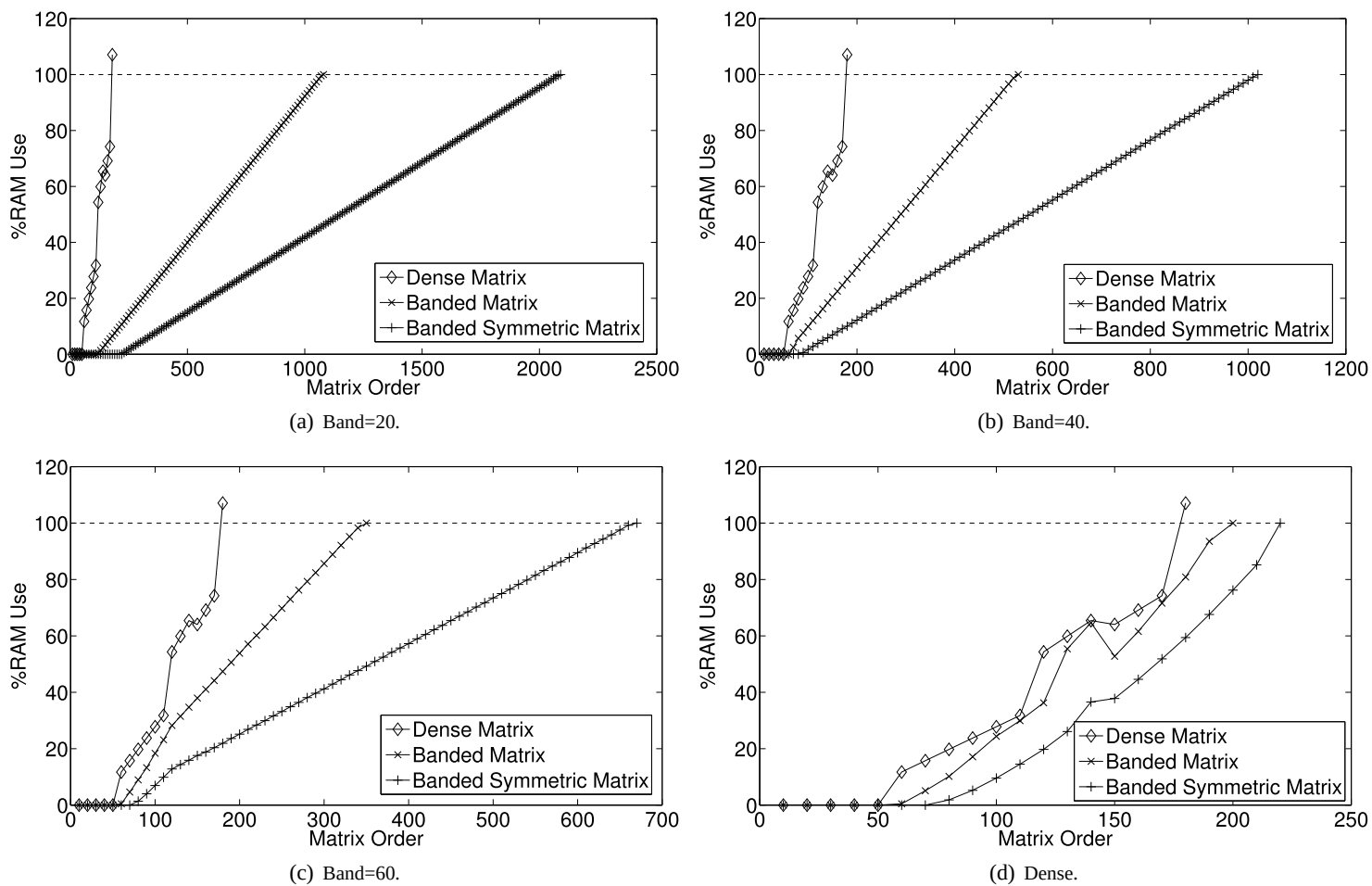


Figure 3.19.: RAM use.

storing a dense matrix, the maximum matrix order can be extended from 120 to almost 1000 in the banded case and 2000 in the symmetric banded case. It should be noted that this bandwidth would, at the maximum, require 39 parallel floating point multiplications, and hence could not be fed using off-chip RAM. As the bandsize increases, though this difference gets smaller, it is still significant.

It is also interesting to see that the difference between the dense and banded case decreases much faster than the difference between the dense and the symmetric case. The reason for this is that the symmetric delay is only a function of N , whereas storing the band instead of implementing this delay is a function of N and P . However, it is clear from the graphs, there are indeed still RAM savings using the banded format as opposed to storing it in the dense format, even where there is a wide band of $M = 60$, as mentioned in section 3.3.1.

Finally it is interesting to see how our approach smoothes all the transitions between the discrete RAM sizes. It is clear in the dense case that there are distinct jumps which are a result of needing a larger RAM size, which is likely to be largely empty, as mentioned in Section 3.3.1, to hold each column of the matrix. In our approach, as we optimise the use of RAMs and shift registers, these sudden jumps generally do not occur. There are however two exceptions to this, the first is for small matrices when it is possible to store the entire matrix using shift registers instead of RAMs, the second is that whenever the parallelism is decreased, there is a sudden increase in slices that could be used as shift registers. However, this second effect could easily be controlled by the choices of X and Y .

Performance

As well as RAM savings, this work can also be used to both achieve greater parallelism, and continue to achieve greater parallelism for higher order matrices. In order to demonstrate this, we again set $P = 10$, then focus on parallelism by setting $Z = 1$, whilst to ensure a fair degree of flexibility to choose the best circuit, we set $Y = 10$ and $X = \min(N, 2M - 1)Y$, which allows for potential for a fully parallel matrix-vector multiplication, provided there are sufficient resources, whilst still maintaining the run-time of the ILP to be only a few seconds. Figure 3.20 compares the maximum performance achievable for a dense matrix using a traditional dot-product approach as in Figure 3.7, our optimisation strategy applied to a dense matrix, and our approach applied to a matrix with a wide band of 80. All designs

were successfully placed and routed to a target frequency of 150 MHz; while it is possible to obtain a higher frequency, such as the designs in Section 3.2, finding the optimum frequency for every point on the graph would take a large amount of time as the circuits consumed almost the entire FPGA and would also distract from our main focus of this section which was to demonstrate how the extra parallelism leads to higher performance, we note that for a single implementation, we would tune this to a higher frequency. In this figure, we have plotted the GFLOPs assuming the circuit to be fully utilised, which is possible in a good design, as we have seen in Section 3.2.

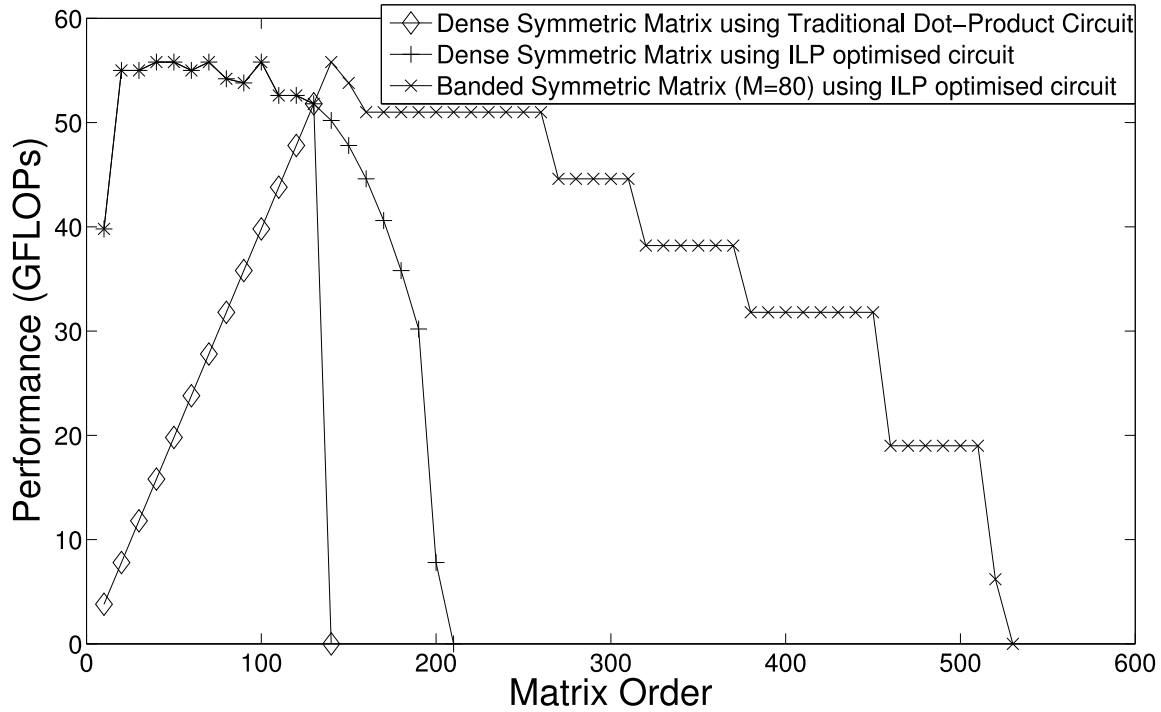


Figure 3.20.: Maximum performance achievable.

Figure 3.20 shows our approach achieves 55 single precision GFLOPs for small matrix orders by performing the dot product of several rows in parallel by making use of almost all the slices available on the device, except for the matrix order of 10 where the maximum parallelism is a fully parallel matrix-vector multiplication, and maintains this high performance for larger matrix orders by performing parallel dot products in stages, in conjunction with the RAM optimisation, before gradually dropping with larger matrix orders due to the RAM requirements allowing less room for parallelism. The slight jumps in the graph which are due to the choices of X and Y that only allow for certain discrete levels of parallelism.

In Section 3.2, the maximum performance of the approach we described achieved was 53 GFLOPs and only when solving the largest matrix order of 145 despite operating at a much higher frequency, and in this section, we noted that the majority of the performance originates from the extensive use and parallelism in the dot-product circuit. In comparison, Figure 3.20 shows that in general, any approach using the traditional full dot-product approach can only approach the performance of our optimised circuit for the maximum matrix order of 170, and cannot scale to larger orders to achieve greater performance due to insufficient memory, shown earlier in Figure 3.19(d).

3.4. Examining the effects of varying the precision used throughout an implementation of the MINRES algorithm

The major limitation of this work is that, as discussed in the introduction of this chapter, all the implementations in this chapter are based upon IEEE 754 single precision floating point. Though the majority of the related work has made similar assumptions, as discussed in Chapter 2, because the accumulation of rounding errors arising from the use of floating point can affect the convergence of the algorithm, and no such analysis has been performed, it is equivocal as to whether this hardware would be of use in a practical situation. Alternatively, suppose such analysis was performed, if the ideal precision was anything but IEEE 754 single precision floating point, the potential performance improvements that can be achieved through changing the precision used throughout the algorithm have also been ignored. In the following sections, we briefly illustrate the importance of precision on convergence and performance.

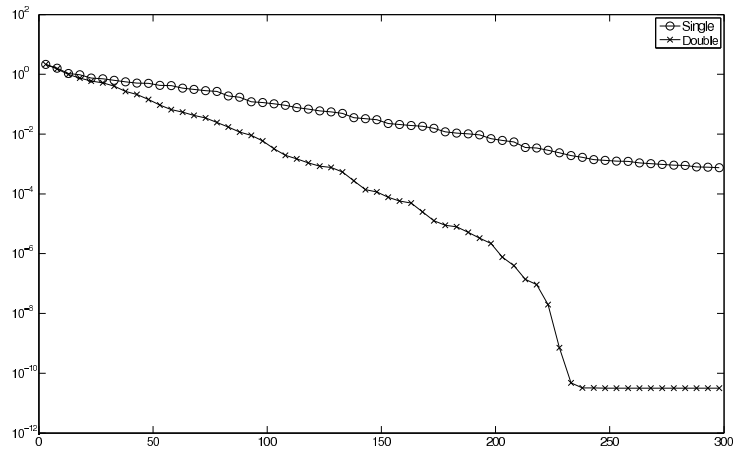
3.4.1. Empirical data on the MINRES algorithm in finite precision

In order to illustrate the effect of the choice of precision on the convergence of the MINRES algorithm, we have conducted a range of simulations which examine the convergence of the MINRES algorithm on different matrices. These matrices were symmetric dense matrices of size 100 generated by randomly sampling with a uniform probability distribution between zero and one for each element to create a random matrix C , and then calculating $A = C^T C$ to make the A matrix symmetric, a b vector generated using a similar distribution for each element, and an initial guess vector set to 0. Figure 3.21

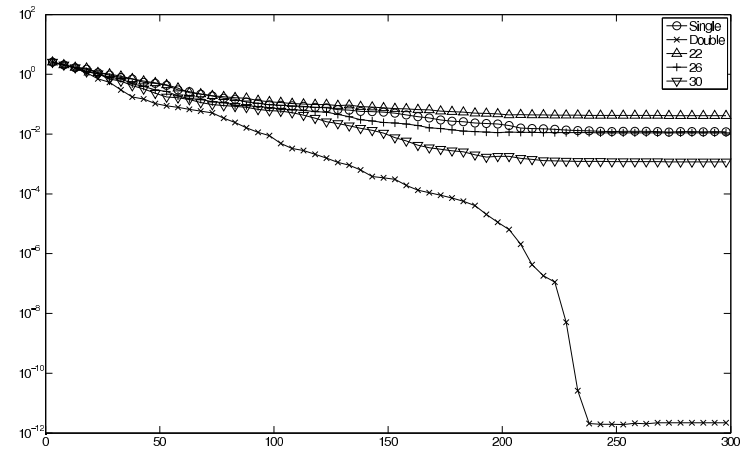
demonstrates how the residual error computed by $\|Ax - b\|_2$ decreases over the number of iterations. Figure 3.21(a) shows the gap in the rate of convergence between single and double precision for a matrix of condition number of order 10^4 , Figure 3.21(b) extends this same model with several different mantissas, illustrating how they occupy space within this gap, while Figure 3.21(c) demonstrates that if the mantissa is too small, the floating point rounding errors can accumulate resulting in a divergence. Finally, Figure 3.21(d) shows that a symmetric matrix with a small condition number of 500, which is a well conditioned matrix, it is possible to converge using much smaller mantissas. These graphs highlight that the choice of precision has a large impact on convergence, that it is desirable not to limit analysis to just double or single precision, because the optimum choice of precision may be somewhere in-between or indeed beyond double precision. It also highlights that the choice of precision will be problem dependent.

Furthermore, while the results above imply that very small mantissas are unlikely to be of interest, if we were to modify the MINRES algorithm by applying a restart strategy, where one restarts the algorithm whenever the true error differs from the actual error by a certain factor, but uses the current x -vector as an initial guess solution, the convergence behaviour can be increased even for small mantissas. This is something that has been studied for the Lanczos algorithm [24, 39, 154, 155], which is a key component in iterative methods including MINRES. Figure 3.22 has the same conditions as in Figure 3.21(a) (random dense symmetric matrix of size 100), but applies a restart strategy. These restarts cause sudden jumps in the graphs, but ensures the system now converges for much smaller mantissas (albeit using significantly more iterations).

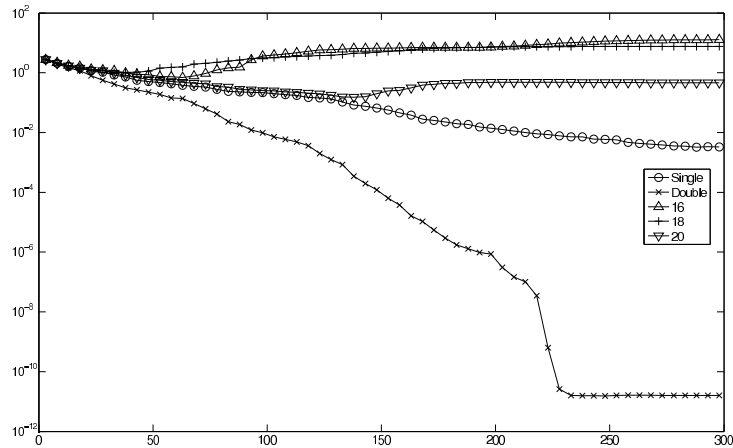
While these graphs highlight that the choice of precision affects the convergence, in general such empirical data would be of little use to choose the optimum precision to use throughout a hardware accelerator. This is because it is too problem dependent given that the order of the matrix, positioning of eigenvalues, and how close the initial guess is to the final solution all effect the convergence, and any non-exhaustive simulation cannot guarantee convergence behaviour over a range of input data. An alternative would be to use theoretical bounds, such as those identified by Greenbaum [64] which highlight bounds on the precision required to ensure convergence for a given matrix condition number, as mentioned in Chapter 2, but calculating these bounds for an arbitrary algorithm requires human ingenu-



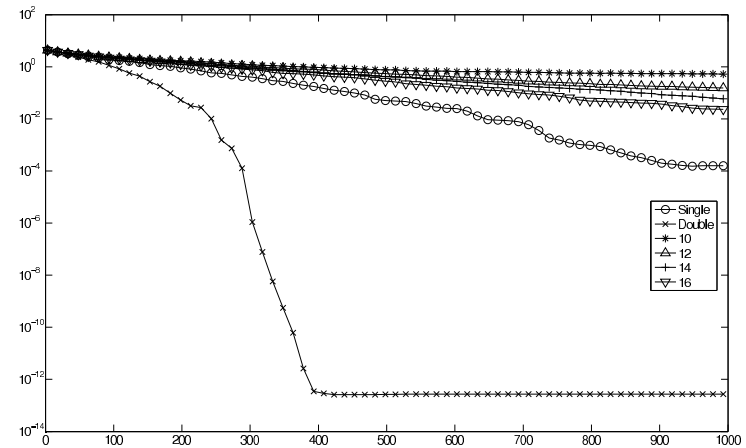
(a) Random Dense Symmetric A Matrix of order = 100 & condition $\approx 10^4$, random b vector, 0 initial guess



(b) Random Dense Symmetric A Matrix of order = 100 & condition $\approx 10^4$, random b vector, 0 initial guess



(c) Random Dense Symmetric A Matrix of order = 100 & condition $\approx 10^4$, random b vector, 0 initial guess



(d) Dense Symmetric A Matrix of order = 100 & condition = 500, random b vector, 0 initial guess

Figure 3.21.: Reduction in residual over iterations for different matrices using various precisions.

ity. A more general method would be to develop an algorithm to calculate bounds on the relative error introduced by the use of finite precision arithmetic on any given computation, and this inspires our work in the subsequent chapters.

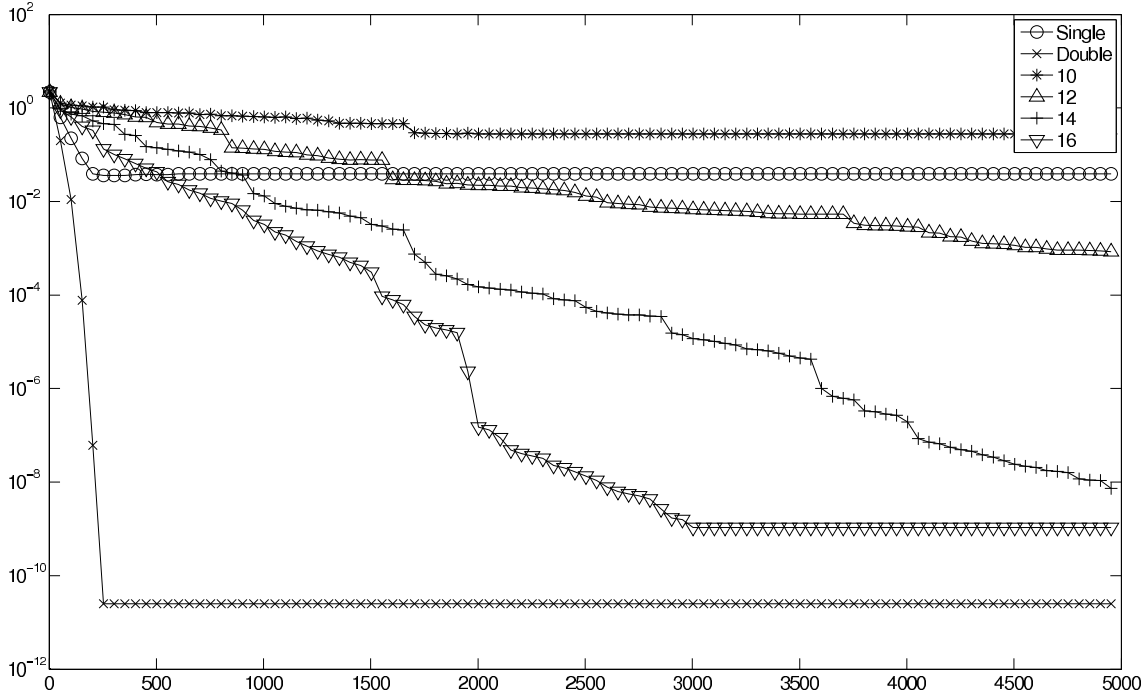


Figure 3.22.: Reduction in residual for random matrix of size 100 for various precisions using restarting.

3.4.2. Estimating the performance impact of varying the precision for a hardware acceleration of MINRES algorithm

We have demonstrated that the choice of precision has a significant impact on the convergence of the algorithm, however to justify why one would wish to create hardware with a lower precision, despite a longer convergence, it is important to study the potential performance gains that can be achieved by using a lower precision.

In order to gain an estimate of the potential performance improvement that could be obtained by tuning the precision, we have adapted our ILP described in Section 3.3.3 to consider variable precision and an approximation of additional silicon area required to compute the MINRES algorithm as described in Section 3.2. In order to estimate the area of the MINRES solver, a simple method would be to add

constants representing the additional floating point operators required in the MINRES solver to the Slice and DSP constraints Figure 3.18. However, as mentioned in Section 3.3.2, if α dot-products are performed in parallel, then in order to avoid pipeline stalls, it is necessary to also parallelise the vector operations in the MINRES accelerator. This constraint can also easily be incorporated into the ILP by adding a constant representing the area for the floating point operators on vector operations and multiplying this by α in the Slice and DSP constraints. Equations (3.8) and (3.9) represent these new constraints, where K_6 to K_9 are the required slices and DSPs for the scalar and vector operations. We also note that performing partial dot-products in parallel in the matrix-vector multiplication circuit affects the latency of the problem, so we set P to ensure that there is sufficient RAM to store the number of required problems to keep this circuit in operation, using the same approach to before where we match the latency of an iteration of the MINRES solver with the time to compute the matrix-vector multiplication of P problems. After these changes, the latency is now given by equation (3.7), where C_1 and C_2 will change according to the mantissa size, and the time to input P problems into the circuit will now be given by $PN\frac{\alpha}{\beta}$. Furthermore, even though the chosen number of parallel dot-products affects P , because there is a distinct number of possible configurations chosen by X and Y , we can calculate the exact value for P for each configuration *a priori* and then use the same indicator variables that previously selected the value of β from the potential configurations in Section 3.3.3 to choose a value for $\beta \times P$.

$$\text{Total Latency (cycles)} = 3N\frac{\alpha}{\beta} + C_1 \lceil \log_2 N\frac{\alpha}{\beta} \rceil + C_2. \quad (3.7)$$

$$K_1 \sum_{j=1}^{\alpha} (\tau_{1ij} + \tau_{2ij}) + K_2\iota + K_3\kappa + K_6\alpha + K_7 \leq S \quad (\text{Available Slice Constraints}) \quad (3.8)$$

$$K_4\iota + K_5\kappa + K_8\alpha + K_9 \leq D \quad (\text{Available DSP Constraints}) \quad (3.9)$$

Finally, we simply adjust the constants for all of the floating point operators (K_1 to K_9), according to the precision used, as well as the constant B , for the RAM use will vary according to the precision, and re-run the ILP to gain performance estimates in variable precision.

Using the modified ILP, we have generated Figure 3.23 which estimates the potential performance

that can be obtained for a MINRES implementation using mantissa widths of 8, 16, 24 (corresponding to IEEE Single precision), 32, 40, 48, and 53 (corresponding to IEEE double precision). The reason we have chosen to approximate the performance is that the time to place and route every design and maximise the operating frequency would be very large, indeed in Section 3.3.4 we simply targetted 150MHz instead of finding the maximum frequency in all cases, and the aim of this section is solely to estimate the potential performance gains, as opposed to actually creating the optimal design. To this end, the graphs in Figure 3.23 are based on the assumption that 150MHz can be achieved. This will estimate a lower bound for the performance of any precision less than single precision, because a frequency of above 150MHz was below that achieved in the MINRES solver in Section 3.2.1 (this leads to a slightly lower performance estimate than that achieved in Section 3.2) and was verified for the optimised matrix-vector multiplication circuit in Section 3.3.4 in single precision, and components using less precision are typically simpler and operate at a higher frequency [157]. In contrast, it is unclear whether the designs using a larger precision than single precision could achieve this frequency, but this still provides us the ability to estimate potential performance. Even with this relaxation with regard to the potential frequency, Figure 3.23 clearly demonstrates that with lower precision, we can achieve a higher performance and maintain this increased performance to much larger matrices. This is due to the smaller size of floating point operators using less precision and the reduced RAM requirements to store data using a reduced precision. We do note that in contrast to Figure 3.20, we do not obtain maximum performance for very small order matrices, this is because the number of problems that must be stored in the pipeline is much higher, as seen in Figure 3.3, and this requires more memory to store the additional elements, which restricts the potential parallelism of the accelerator.

3.5. Summary

In summary, this chapter has described how it is possible to achieve a high sustained performance of 53 GFLOPs for MINRES, and has also shown that because the majority of this performance is a function of the matrix-vector multiplication circuit, by applying a few simple changes to this circuit architecture and taking advantage of convex optimisation techniques, we can achieve even greater performance that

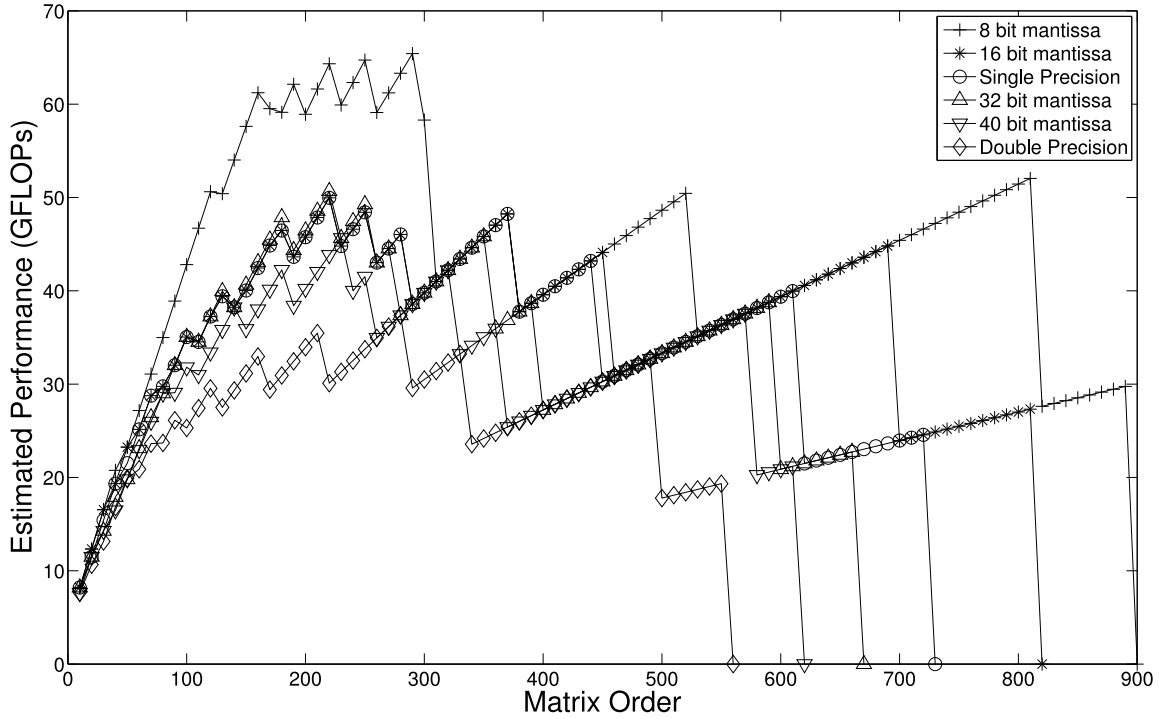


Figure 3.23.: Estimated potential performance using ILP for MINRES. These figures are based upon a circuit running at 150 MHz, which we believe to be a pessimistic operating frequency, and below that used in Section 3.2, but one which all implementations should be able to achieve.

could be sustained to larger matrices. We also note that the matrix-vector multiplication circuit we have described is highly parameterisable and may be of use in a wide range of applications, especially problems which include banded and symmetric matrix structure, as well as being incorporated in the MINRES solver or other hardware accelerators for iterative methods.

Finally, this chapter has illustrated that the precision used throughout the hardware implementation has a significant impact both on the convergence of the MINRES algorithm and of the potential performance that could be achieved. However, these effects are not specific to MINRES, in general, the accumulation of floating point errors will affect whether an algorithm satisfies a design criterion, while the silicon area savings of using less precision will always allow room for additional parallelisation resulting in greater hardware acceleration. In the subsequent chapters of this thesis, we attempt to generate new analytical techniques to compute bounds on the range or relative error to help a designer take advantage of these potential performance advantages.

Chapter 4

Bounding variable values and round-off effects using Handelman representations

In the final section of the previous chapter, we illustrated that the choice of precision has a large impact on properties such as the convergence of an algorithm and the potential performance of the hardware. However, in order to take advantage of the potential performance benefits, we need a tool to assist in making an informed choice of precision. This choice should be made according to some design criterion; such a design criterion may be a range specification or a bound on the worst case computational error induced so as to ensure convergence or a given safety margin.

Unfortunately, as we have highlighted in Chapter 2, there is currently a limited number of tools available to calculate such bounds. As a brief reminder of these issues, simulation-based methods cannot guarantee the given error estimate while the existing analytical tools tradeoff quality of bounds for execution time, with the existing methods currently lying at the extremes of this spectrum. The most sophisticated class of analytical methods that can calculate bounds in a tractable time fall under the name of Taylor forms [115]. Generally speaking, these methods construct polynomials, with the aid of various approximations, that represent the range of every variable in an algorithm and then calculate bounds on the final polynomial using interval arithmetic. We note here that the difference between the Taylor forms are the approximations used to control the size of each polynomial and ensure that it always bounds the true range. While applying approximations to control the size of the polynomial provide the ability to trade execution time with quality of bounds, issues we will discuss in greater detail in Chap-

ter 5, the use of interval arithmetic in these stages, as well as the final stage of bounding the polynomial T_p representing the value of a variable, limit the quality of bounds Taylor forms can achieve. Once again, this is because interval arithmetic is unable to find the best bounds for any polynomials that contain dependencies [108]. As most algorithms re-use variables, these dependencies will almost always occur, so in this chapter we focus on creating a new approach that can take into account dependency information when bounding the polynomials which represent the range of a variable.

In this chapter, we first detail the notation to be used throughout this and the subsequent chapter in Section 4.1 before we describe how to create a polynomial to represent the range for each variable in the pseudo code of the algorithm using models for floating point or fixed point error in Section 4.2. We then provide a brief background discussion on various results from the fields of convex optimisation and real algebraic geometry which could be used to bound a polynomial in Section 4.3; this both illustrates the need to create a new approach, as well as providing the background theory for our new heuristic which we detail in Section 4.4. Our benchmarks and the methods to quantify the quality of bounds of our approach are then discussed in the antepenultimate section, Section 4.5, before our results are presented in Section 4.6, along with a demonstration of how our new approach could be used to design hardware, and demonstrate the potential hardware savings that could be achieved. We remark at this point that while this new approach could be of value in proving whether it is safe to move from double to single precision for use on alternative hardware accelerators, such as GPUs or general purpose processors, because these devices are largely limited to these two choices of floating point precision, in this chapter we again choose to present results for FPGA architectures. Finally, we summarise this chapter in Section 4.7, and highlight the limitations of the heuristic in this chapter, limitations which are partially addressed in Chapter 5 and partially left for future exploration, which we discuss in Chapter 6.

The main original contributions of this chapter are as follows:

- the description of a new method, based upon a result from real algebraic geometry, to find provable bounds for any variables within a sizeable computational kernel, given input data ranges and a precision specification,
- a demonstration of the use of our new approach to design hardware with a reduced global word-

length.

4.1. Notation

To formalise the discussion of the method, some simple notation is used. We consider polynomials in n variables $\delta_1, \delta_2, \dots, \delta_n$. We use the notation δ^λ for a term, which is a product of the variables raised to some integer powers (4.1), where λ is a vector collecting the exponents (4.2). We denote by $|\lambda|$ the *degree* of the term, given by (4.3).

$$\delta^\lambda = \delta_1^{\lambda_1} \delta_2^{\lambda_2} \dots \delta_n^{\lambda_n}. \quad (4.1)$$

$$\lambda = (\lambda_1, \dots, \lambda_n), \text{ where } \lambda_i \in \mathbb{N}. \quad (4.2)$$

$$|\lambda| = \lambda_1 + \dots + \lambda_n. \quad (4.3)$$

A monomial is defined as a term multiplied by some real coefficient, *i.e.* $c\delta^\lambda$, a polynomial is the sum of one or more monomials, and a rational function as a function consisting of a numerator and denominator polynomial.

Finally, to allow a formal description in the following sections, in this work the terms in a polynomial are ordered. $\delta^\mu < \delta^\lambda$ denotes that δ^μ precedes δ^λ , according to degree lexicographical order, as described in (4.4) [2].

$$\delta^\mu < \delta^\lambda \Leftrightarrow \begin{cases} |\mu| < |\lambda|, \\ \text{or} \\ |\mu| = |\lambda| \text{ and } \exists i(\mu_i < \lambda_i \text{ and } \forall j < i(\mu_j = \lambda_j)) \end{cases} \quad (4.4)$$

4.2. Creating a polynomial representation of potential range

Throughout the majority of this chapter, especially when evaluating our approach in Section 4.6, we generally focus on optimising floating-point hardware. This is partially due to the relative lack of existing work discussing word-length optimisation for floating-point designs in comparison to fixed-point,

and partially due to the recent trends showing that floating-point designs are becoming highly efficient in hardware [146], and the growing collection of publications of floating-point custom hardware implementations [11,95]. We note, however, that the approach we describe in Section 4.4 is equally applicable to word-length optimisation for fixed-point designs by using a different model of error discussed in Section 4.2.2. In the remainder of this section, we describe how to construct a polynomial representing the potential range of a variable for both models of error.

4.2.1. Floating point

It can be shown that for a real value x , the closest radix-2 floating-point approximation $\hat{x}$ of x can be expressed as in (4.5) [73], where η is the number of mantissa bits used (referred to as the precision), provided there is no overflow or underflow (note our approach could easily be extended to support any radix by changing this equation). To ensure there is no overflow, it is necessary to perform range analysis, taking into account errors introduced by floating point precision, and use the results to choose the number of bits required for the exponent; this can be done using the approach we describe in Section 4.4. Underflow depends on the chosen exponent and only occurs as a variable nears zero, as such, by performing range analysis taking into account errors for every variable, we can determine whether underflow is possible, and if so, we can add an additional error term bounding this error. Once we have ensured there is no overflow and any underflow is bounded, using this model it is similarly possible to specify that the radix-2 floating-point result of any scalar operation ($\odot \in \{+, -, *, /\}$) is bounded as in (4.6), provided the exponent is sufficiently large to span the range of the result. Operations complying with IEEE standard arithmetic exhibit this behaviour.

$$\hat{x} = x(1 + \delta_1) \quad (|\delta_1| \leq \Delta, \text{ where } \Delta = 2^{-\eta}). \quad (4.5)$$

$$\widehat{x \odot y} = (x \odot y)(1 + \delta_1). \quad (4.6)$$

We can apply this model of floating-point error on the result of every computation throughout an algorithm such that every output variable can be represented by a single polynomial in all the error

variables, as shown for a simple example in Table 4.1.

Table 4.1.: Construction of polynomials with floating point error.

x, y are inputs	
Δ is the error bound determined by the precision, so that $ \delta_i \leq \Delta$	
Pseudo code	Polynomial representation of variable value
$a = x*y;$	$a = xy(1 + \delta_1)$
$b = a*a;$	$b = (xy(1 + \delta_1))^2(1 + \delta_2)$
$c = b-a;$	$c = [(xy(1 + \delta_1))^2(1 + \delta_2) - xy(1 + \delta_1)](1 + \delta_3)$

4.2.2. Fixed point

When using fixed point, provided there is no overflow, the errors introduced are a result of truncation. As with floating point error, to guard against overflow errors, we must perform range analysis, taking into account errors introduced by the use of fixed point, to choose the minimum number of bits to reach the maximum potential value for any given variable. After this is performed, then assuming the fixed point operators perform the operation and find the correct result before truncating the value to the desired fixed point precision, the maximum truncation error will always be limited to one unit in the last place. If we denote the maximum value for the chosen number system as 2^X , the bound for any fixed point number is given by (4.7), and similarly as for floating point, the result of any scalar operation ($\odot \in \{+, -, *, /\}$) is bounded as in (4.8).

$$\hat{x} = x + \delta_1 \quad (|\delta_1| \leq \Delta, \text{ where } \Delta = 2^{X-\eta}). \quad (4.7)$$

$$\widehat{x \odot y} = (x \odot y) + \delta_1. \quad (4.8)$$

The main difference between the fixed point model of error and the floating point model of error is that the fixed point model is additive, whereas the floating point model is multiplicative; this will typically result in smaller polynomials which are easier to bound, as we will discuss in Section 4.6.4. Table 4.2 demonstrate the models created by the fixed point model of error for the same pseudo code as the example in Table 4.1 so as to compare and contrast these models

Table 4.2.: Construction of polynomials with fixed point error.

x, y are inputs Δ is the error bound determined by the position of the fixed point ($\Delta = 2^{X-\eta}$), so that $ \delta_i \leq \Delta$	
Pseudo code	Polynomial representation of variable value
$a = x*y;$	$a = xy + \delta_1$
$b = a*a;$	$b = (xy + \delta_1)^2 + \delta_2$
$c = b-a;$	$c = [(xy + \delta_1)^2 + \delta_2] + \delta_3$

4.3. Calculating bounds of a multivariate polynomial

Given a multivariate polynomial representing the value of a variable, as in Tables 4.1 or 4.2, $f(\delta)$, we want to find $\gamma_{lower} = \inf_{|\delta_i| \leq \Delta} f(\delta)$, the lower bound on the variable or function of intent, and $\gamma_{upper} = \sup_{|\delta_i| \leq \Delta} f(\delta)$. Unfortunately this is a non-convex optimisation problem, which is NP-hard, meaning traditional approaches are unsuitable. For example, calculus style approaches towards bounding a polynomial of order ρ by searching for turning points are unsuitable because finding the solutions for $f'(\delta) = 0$ for a general multivariate polynomial is also NP-hard. As a result, we focus on finding a computationally tractable lower bound $\hat{\gamma}_{lower} \leq \gamma_{lower}$ and upper bound $\hat{\gamma}_{upper} \geq \gamma_{upper}$ such that $\gamma_{lower} - \hat{\gamma}_{lower}$ and $\hat{\gamma}_{upper} - \gamma_{upper}$ are as small as possible. Since the variables in these polynomials are bounded $|\delta_i| \leq \Delta$, this problem falls into the broad field of global optimisation over a bounded box [6], so in this section we provide some background theory into methods used to compute such bounds so as to emphasise the need to develop our new heuristic which we discuss in the following section.

4.3.1. Bernstein expansion

The use of the Bernstein expansion to calculate bounds on polynomials is a topic that has gained considerable momentum in the *reliable computing* community. To gain a basic understanding of how these work, it is best to first consider the univariate case, where Bernstein Polynomials of the form (4.9) are used to construct a function called the Bernstein expansion, of the form (4.10), to approximate the desired function $f(\delta)$. The main computation of this approach is the computation of the so-called Bernstein Coefficients, (4.11), by a method such as the de Casteljau algorithm [54]. This can then be extended for multivariate polynomials by computing products of the univariate Bernstein polynomials [58].

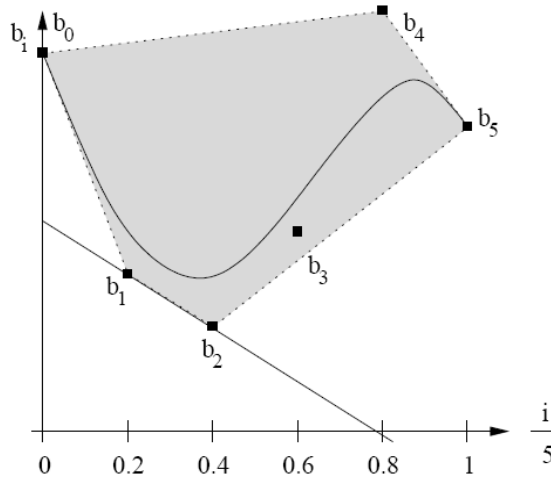


Figure 4.1.: Convex hull property of Bernstein coefficients. Figure taken from [59].

$$B_i(x) = \binom{l}{i} x^i (1-x)^{l-i}, i = 0, \dots, l \quad (4.9)$$

$$p(x) = \sum_{i=0}^l b_i B^i(x). \quad (4.10)$$

$$b_i = \sum_{j=0}^i \frac{\binom{i}{j}}{\binom{l}{j}} a_j, i = 1, \dots, l. \quad (4.11)$$

The main beneficial property of this approach is that these Bernstein coefficients satisfy the convex hull property for this polynomial. Figure 4.1, which is a direct reference from [59], demonstrates how these coefficients can be used to construct the convex hull for a polynomial of fifth degree. The convex hull property means that these coefficients will always bound the polynomial and can therefore be used to calculate worst case bounds, or even approximate lower bound functions [58], though this is beyond the requirements of our problem. The other beneficial property is that it has been shown that these bounds converge quadratically to the lower and upper bounds when interval splitting is performed for the univariate case, a property that is expected to hold for the multivariate case [58].

However, there are several problems with this approach. Firstly, computing the Bernstein coefficients has been shown to have computational complexity of $O(n\rho^{n+1})$ for a polynomial of order ρ consisting

of n variables [59], and while efforts have gone into reducing this computational complexity for sparse polynomials [136], sparse polynomials will not necessarily arise in our problem domain. Secondly, while the convex hull property ensures that the Bernstein coefficients correctly bound the result, there is no indication of the tightness of this bound and it is necessary to take advantage of the convergence of the bounds by interval splitting to ensure it is tight. However, as we mentioned in Chapter 2, relying on interval splitting will cause the computational complexity to calculate the bounds to grow exponentially again.

4.3.2. Lagrangian duality

A more traditional approach to calculate lower $\hat{\gamma}_{lower}$ and upper bounds $\hat{\gamma}_{upper}$ for a multivariate polynomial $f(\delta)$ over a bounded box is to formulate it as a nonconvex optimisation problem, such as (4.12), and then to take advantage of Lagrangian duality to simplify this to a convex optimisation problem. In this approach, one augments the constraints on the variables (δ_i) with the objective function ($f(\delta)$) to form a single function, given by (4.13), which is known as the Lagrangian dual function, with the property that any positive value of ϕ for the Lagrangian dual function provides a lower bound to the initial problem, *i.e.* $g(\phi) \leq f(\delta)$. Furthermore, as the Lagrangian dual function is a concave function, it is possible to maximise $g(\phi)$ using convex optimisation techniques to compute the tightest possible bound for the approximation [19].

$$\begin{aligned} & \min \quad f(\delta) \\ \text{subject to: } & \forall i, \quad \Delta - \delta_i \geq 0 \\ & \forall i, \quad \Delta + \delta_i \geq 0 \end{aligned} \tag{4.12}$$

$$g(\phi) = \inf_{\delta} \left(f(\delta) + \sum_{i=1}^n \phi_i(\Delta - \delta_i) + \sum_{i=n+1}^{2n} \phi_i(\Delta + \delta_i) \right) \tag{4.13}$$

Unfortunately, it is often difficult to compute the infimum of a general multivariate function so as to create the dual function; indeed this problem is only easier than the original formulation given in (4.12) if it is possible to choose ϕ to remove monomials from the polynomial $f(\delta)$. Furthermore, even if it is possible to compute a useful Lagrangian function, while convex optimisation tools are rich, the problem with this approach is that this approximation potentially creates a gap between the optimal solution to the primal and the dual problem, *i.e.* $f(\delta) - g(\phi) \geq 0$ (in the case $f(\delta) < 0$, the infimum is given by $-\infty$ (which is not a useful lower bound), and as with Bernstein polynomials, there is no indication of the tightness of this bound.

4.3.3. Theorems of alternatives

Theorems of alternatives are an area of real algebraic geometry which has grown significantly since the discovery of Farkas' lemma in 1902 [19]. The high-level concept of these approaches is to construct an alternate system for any given system of constraints such that only one of these two systems is feasible. Provided the alternate system is easier to solve than the initial system, by finding a solution to the second system, we obtain a certificate of infeasibility to the initial system. The reason it is of interest in calculating bounds is that if we can use a theorem of alternatives to find a certificate which shows $f(\delta) < \hat{\gamma}_{lower}$ is infeasible, then it means that $f(\delta) \geq \hat{\gamma}_{lower}$, and gives us a lower bound. We could apply a similar analogy to calculate an upper bound.

A classical example of a theorem of alternatives is a simple extension based on Lagrangian duality, as described in Section 4.3.2. If we set the objective function to 0, and move the condition for the upper or lower bound to one of the constraints, as shown in (4.14) the solution will either be 0 if it is feasible, otherwise it is ∞ . With this problem, the dual function is given by (4.15). We note that if $g(\phi) \leq 0$, the maximum is 0, however, if we can show $g(\phi) > 0$, then because of the additional variable ϕ_0 , for any constant c , $g(c\phi) = cg(\phi)$, meaning the maximum in this case is ∞ . We can therefore say the bounds for maximum value of $g(\phi)$, denoted $g(\phi^*)$, are given by (4.16). Finally, by duality, $g(\phi) \leq f(\delta)$, meaning that if $g(\phi) > 0$ is feasible, $\infty \leq g(\phi) \leq f(\delta)$, which provides a certificate to show (4.14) is infeasible [19].

$$\begin{aligned}
 & \min \quad 0 \\
 & \text{subject to: } \hat{\gamma}_{lower} - f(\boldsymbol{\delta}) \geq 0 \\
 & \quad \forall i, \quad \Delta - \delta_i \geq 0 \\
 & \quad \forall i, \quad \Delta + \delta_i \geq 0
 \end{aligned} \tag{4.14}$$

$$g(\phi) = \inf_{\boldsymbol{\delta}} \phi_0(\gamma_{lower} - f(\boldsymbol{\delta})) + \sum_{i=1}^n \phi_i(\Delta - \delta_i) + \sum_{i=n+1}^{2n} \phi_i(\Delta + \delta_i) \tag{4.15}$$

$$g(\phi^*) = \begin{cases} \infty, & \text{if } g(\phi) > 0 \text{ is feasible.} \\ 0, & \text{otherwise.} \end{cases} \tag{4.16}$$

There are however two problems with this approach. The first is the difficulty to calculate all the values of ϕ to satisfy $g(\phi) > 0$. The second is that using Lagrangian duality, in general, as the initial problem is nonconvex, the duality gap still exists, and similar to before, this means that even if we find a certificate $g(\phi) > 0$ for a given $\hat{\gamma}_{lower}$ to show that (4.14) is infeasible, there could potentially exist a certificate which can prove a tighter bound. However, there do exist other theorems of alternatives from real algebraic geometry that can be used to produce so-called strong certificates where the duality gap is zero; we discuss two such theorems in the following subsections.

Positivstellensatz refutations

The Positivstellensatz is one theorem that can be used to find strong certificates; the proof and complete definition for a system consisting of linear inequalities, equalities and disequalities can be found in [126], however, Theorem 1 presents a simplified version that is more relevant to our problem where we are only interested in systems of linear inequalities, as in (4.12).

Theorem 1 [126]. *The set $S_c = \{x \in \mathbb{R}^n | g_i(x) \geq 0\}$ is empty if and only if there exists a polynomial p*

in the cone (of the form (4.17)) over the compact set of linear inequalities $g_i(x) \geq 0$ such that $p = 0$.

$$p = \sum_{\alpha \in \mathbb{N}^n} k_{\alpha} \prod_{i=1}^n g_i^{\alpha_i}(x), \quad (4.17)$$

where each $k_{\alpha}(x)$ is a non-negative polynomial

While it initially appears difficult to use this theorem to find refutations, Parillo demonstrated that it is possible to use this theory and search for bounded degree Positivstellensatz refutations using semi-definite programming (SDP) to find sum of squares values for each k_{α} , which are by definition non-negative [126]. The reason that the search is performed over a bounded degree is that the product of products of all the non-negative inequalities could result in a large semi-definite program which will take a long time to solve, and limiting the degree provides a control over the execution time at the cost of reducing the quality of bounds. We remark that this type of control was unavailable using the methods described in Sections 4.3.1 or 4.3.2. However, while this method is very powerful, the number of coefficients in the semi-definite program limited to degree ρ is given by $\binom{\rho+2n}{2n}$ meaning that even for modest degrees, this program becomes too large to solve using this technique. Furthermore, even modern SDP solvers are unable to solve large problems in a short execution time [107].

Handelman representations

A simplification of the Positivstellensatz which focused on looking for certificates of infeasibility for a system which only consists of polynomial inequalities was discovered by Handelman.

Theorem 2 [67]. *A polynomial $p(x)$ is non-negative over the compact set of linear inequalities $g_i \geq 0$, i.e. $S_c = \{x \in \mathbb{R}^n | g_i(x) \geq 0\}$, if and only if p has a Handelman representation of the form (4.18).*

$$p = \sum_{\alpha \in \mathbb{N}^n} c_{\alpha} \prod_{i=1}^n g_i^{\alpha_i}, \quad (4.18)$$

where each c_{α} is a non-negative constant

and $\mathbb{N}$ is the set of natural numbers.

In a similar fashion to searching for Positivstellensatz refutations, using this theorem, it is possible to find certificates of non-negativity of a polynomial by finding a Handelman representation for it. However, this search is much simpler because instead of searching for non-negative polynomials $k_{\alpha}(x)$, it is only searching for non-negative coefficients c_{α} , meaning these representations can be found by using linear programming to match the corresponding monomial in the desired polynomial f . The linear program would then be able to calculate all the values for the variables c_{α} , and the desired bound γ . However, it is important to note at this stage that using this method, that while it is possible to find exact Handelman representations, it is impossible to guarantee the global minimum will be found, because Handelman representations are only guaranteed to converge to the global optimum as the maximum order tends to infinity [91]. We demonstrate how linear programming can be used to calculate bounds using Handelman representations, as well as the how the potential need for high degree Handelman representations increases the difficulty to compute the optimum bounds in the following example.

Example of computing Handelman representations with linear programming Calculate the lower and upper bounds of the polynomial $f(\delta_1) = \delta_1^2$ where $|\delta_1| < 1$. We know trivially that the optimum bounds should be $[0; 1]$

To calculate the upper bound using Handelman representations, we need to find a Handelman representation p for the polynomial $\hat{\gamma}_{upper} - \delta_1^2$ to show it is positive. To calculate this Handelman representation, we can apply linear programming, for this can find the optimum bound for a Handelman representation limited to given degree. For this example, we know the minimum degree for the Handelman representation is 2, because the degree of $f(\delta_1)$ is 2. The linear program for this problem is then given by (4.19). Solving this linear program sets the constants c_0 to c_4 to 0, $c_4 = 1$ and $\hat{\gamma}_{upper} = 1$, which we note is the ideal upper bound.

$$\begin{aligned}
 & \min \hat{\gamma}_{upper} \\
 & \text{subject to: } \Delta - \delta_1 \geq 0 \\
 & \Delta + \delta_1 \geq 0 \\
 & c_0 + c_1(1 - \delta_1) + c_2(1 + \delta_1) + c_3(1 - \delta_1)^2 + c_4(1 + \delta_1)^2 \\
 & \quad + c_5(1 - \delta_1)(1 + \delta_1) = \hat{\gamma}_{upper} - \delta_1^2.
 \end{aligned} \tag{4.19}$$

To adopt a similar approach to calculate the lower bound using Handelman representations, we need to find a Handelman representation p for the polynomial $\delta_1^2 - \hat{\gamma}_{lower}$ to show it is positive. The linear program for this problem is given by (4.20), and this sets the constants $c_0, c_1, c_2, c_5 = 0$, $c_3, c_4 = 1/2$ and $\hat{\gamma}_{lower} = -1$, however we note that in this case, this is not the best lower bound.

$$\begin{aligned}
 & \min -\hat{\gamma}_{lower} \\
 & \text{subject to: } \Delta - \delta_1 \geq 0 \\
 & \Delta + \delta_1 \geq 0 \\
 & c_0 + c_1(1 - \delta_1) + c_2(1 + \delta_1) + c_3(1 - \delta_1)^2 + c_4(1 + \delta_1)^2 \\
 & \quad + c_5(1 - \delta_1)(1 + \delta_1) = \delta_1^2 - \hat{\gamma}_{lower}
 \end{aligned} \tag{4.20}$$

To obtain a better approximation, it is necessary to create Handelman representations of greater degrees, in Table 4.3, we demonstrate how the bounds tend to the optimum lower bound as the maximum order increases. In contrast, we note that if we were to search for Positivstellensatz refutations instead of Handelman representations, we could easily compute an exact lower bound because we can use the fact δ_1^2 is itself a non-negative polynomial.

While the fact we cannot always find exact Handelman representations is not ideal, the theory does ensure that we always correctly bound the polynomial, and in many cases it is possible to find exact Handelman representations to obtain the tightest bounds. Furthermore, we could use simple techniques

Table 4.3.: Using linear programming to compute Handelman representations to find bounds of $f(\delta_1) = \delta_1^2$, where $|\delta_1| < 1$.

Degree of Handelman Representation	Lower Bound ¹ ($\hat{\gamma}_{lower}$)	Execution Time (s)
2	-1	0.2
4	-0.334	0.5
6	-0.2	1
8	-0.143	2
10	-0.112	3
12	-0.091	4
14	-0.077	7
16	-0.067	10
18	-0.059	13.5
20	-0.053	18

¹ Rounded downwards to 3 decimal places such that these are valid lower bounds.

to improve our bounds, such as adding an extra constraint $\delta_1^2 > 0$ in the above example to obtain the optimal bounds. However, the greater problem is that even though linear program solvers are generally more efficient than semi-definite program solvers, the linear programming approach suffers from the same problem in searching for Positivstellensatz refutations in that it is not scalable. Table 4.3 also shows how the execution time scales for the simple example program grows with increasing degree. This execution time would grow significantly faster for a multivariate polynomial. As before, this is because a polynomial f of order ρ consisting of n variables, the number constraints is $\binom{\rho+n}{n}$ and variables is $\binom{\rho+2n}{2n}$. Therefore, in order to ensure we get a result in tractable time, we must first restrict the maximum order for the Handelman representation to some value ρ . Furthermore, because one can only choose coefficients, whilst choosing this value of ρ will trade execution for quality of bounds, the value of ρ must be greater than or equal to the degree of f in order to create such a representation. Clearly, whilst this is efficient for small problems, the size of the linear program quickly grows too large for existing linear programming tools. For example, consider a problem consisting of approximately $n = 30$ variables where the maximum order of f is restricted to $\rho = 6$. This would consist of almost 2 million constraints and over 90 million variables, which is unsolvable using current linear programming solvers.

4.4. Our heuristic to bound multivariate polynomials

The aim of this work is, for a given precision, to find bounds on the value of any chosen variable within an algorithm, and hence bound the worst case computational error induced. Our approach consists of several stages. The first stage involves a simple compiler which takes input code and applies this model of floating-point error on the result of every computation throughout an algorithm such that every output variable can be represented by a single polynomial in all the error variables. In this chapter, we focus on straight-line code algorithms consisting of $\{+, -, *, /\}$ operators, meaning that we currently unroll any loops, which is a reasonable approach for real-time computation, where the loop bounds are known. Any conditional statements do not directly affect our bounds procedure if they do not operate on any of the rounded variables. This polynomial is then simplified into a canonical form, from which bounds for the extrema of the polynomial can be found via a heuristic. This heuristic is inspired by Theorems of alternatives, for they provide the possibility of finding the tightest bounds, and within this field, we have chosen to focus on finding a creating a new heuristic which searches for Handelman representations. While we leave open the possibility of enhancing this heuristic to search for Positivstellensatz refutations, in this chapter, we have chosen to focus on Handelman representations because of the simplicity of searching for single coefficients, as opposed to positive polynomials. An optional final stage extends the approach for polynomials to rational functions, allowing all algorithms consisting of the basic operators $\{+, -, *, /\}$ to be automatically analysed.

To address the scalability issues, we design our heuristic which initialises all the coefficients c_α to be zero, then dynamically chooses which coefficients to alter to create the Handelman representation; this is significantly different to the linear programming approach which optimises every possible c_α up to a given degree. To achieve this, in contrast to constructing an optimisation problem as in (4.19), our approach searches for Handelman representations directly. We attempt to find lower and upper bounds to satisfy $\hat{\gamma}_{lower} \leq f(\delta) \leq \hat{\gamma}_{upper}$ by considering that we are trying to show that the functions $f(\delta) - \hat{\gamma}_{lower}$ and $\hat{\gamma}_{upper} - f(\delta)$ are non-negative over the compact set of inequalities specifying the bounds on δ given in (4.21), which by Theorem 2 is equivalent to showing $f(\delta) - \hat{\gamma}_{lower}$ has a Handelman representation of the form (4.22), or similarly satisfying (4.23) for the upper bound.

$$S_c = \{\delta \in \mathbb{R}^n \mid \forall i (\Delta - \delta_i \geq 0) \wedge (\Delta + \delta_i \geq 0)\}. \quad (4.21)$$

$$f(\delta) - \hat{\gamma}_{lower} = \sum_{\alpha, \beta \in \mathbb{N}^n \times \mathbb{N}^n} c_{\alpha, \beta} \prod_{i=1}^n (\Delta - \delta_i)^{\alpha_i} (\Delta + \delta_i)^{\beta_i}. \quad (4.22)$$

$$\hat{\gamma}_{upper} - f(\delta) = \sum_{\alpha, \beta \in \mathbb{N}^n \times \mathbb{N}^n} c_{\alpha, \beta} \prod_{i=1}^n (\Delta - \delta_i)^{\alpha_i} (\Delta + \delta_i)^{\beta_i}. \quad (4.23)$$

Example

In order to demonstrate the use of this theory, we will consider the function $f(\delta_1) = \delta_1^2 - \delta_1$ over the set $|\delta_1| \leq 1/2$. For this simple function of one variable, we can first derive lower and upper bounds using familiar calculus arguments. We will then demonstrate that IA is unable to calculate the same bounds, before finally showing that Handelman representations can be used to prove the ideal bounds.

Using calculus to calculate bounds of $f(\delta_1) = \delta_1^2 - \delta_1$, we know that because this is a convex function, the minimum will lie where the derivative $f'(\delta_1) = 0$ and the maximum will lie at one of the extremes of the range of δ_1 . Thus by differentiating $f(\delta_1)$ to get $f'(\delta_1) = 2\delta_1 - 1$, we find the minimum lies at $\delta_1 = 1/2$, leaving the maximum to be where $\delta_1 = -1/2$; this gives the range of $f(\delta_1)$ to be $[-1/4, 3/4]$. In comparison, interval arithmetic is unable to calculate the optimal lower bound, as shown in (4.24).

$$\begin{aligned} \delta_1 &\in [-1/2, 1/2] \\ \Rightarrow \delta_1^2 &\in [0, 1/4] \\ \Rightarrow \delta_1^2 - \delta_1 &\in [-1/2, 3/4] \end{aligned} \quad (4.24)$$

To find bounds using the theory presented in this section, we want to search for Handelman representations to satisfy (4.22) and (4.23). Two such Handelman representations are $p_{lower_ghr} = (1/2 - \delta_1)^2$ and $p_{upper_ghr} = (1/2 - \delta_1)(1/2 + \delta_1) + (1/2 + \delta_1)$. After equating these respective functions, as shown in (4.25) and (4.26), we find $\hat{\gamma}_{lower} = -1/4$ and $\hat{\gamma}_{upper} = 3/4$. Finally, by Theorem 2, we now know that $f(\delta_1) - (-1/4) \geq 0$ and $3/4 - f(\delta_1) \geq 0$ or that $-1/4 \leq f(\delta_1) \leq 3/4$, thus recovering the optimal

bounds.

$$\delta_1^2 - \delta_1 - \hat{\gamma}_{lower} = (1/2 - \delta_1)^2 \quad (4.25)$$

$$\delta_1^2 - \delta_1 - \hat{\gamma}_{lower} = 1/4 - \delta_1 + \delta_1^2$$

$$\hat{\gamma}_{lower} = -1/4$$

$$\hat{\gamma}_{upper} - (\delta_1^2 - \delta_1) = (1/2 - \delta_1)(1/2 + \delta_1) + (1/2 + \delta_1) \quad (4.26)$$

$$\hat{\gamma}_{upper} - \delta_1^2 + \delta_1 = 1/4 - \delta_1^2 + 1/2 + \delta_1$$

$$\hat{\gamma}_{upper} = 3/4$$

4.4.1. Generalised version of Handelman representations

For our purposes, to simplify our description when working with multivariate polynomials, it will be easier to work with a generalised version of the Handelman representation that we propose, given in (4.27), which we refer to as a *Generalised Handelman Representation* (GHR). This format enables us to describe the GHR using only the set of vectors $\mu_{i,j}, \alpha_{i,j}, \beta_{i,j}$ and the constant c_j , as we will see in Table 4.4.

Theorem 3. *A polynomial p_{ghr} is non-negative over the compact set S_c , defined in (4.21), if and only if p_{ghr} can be represented by a Generalised Handelman Representation of the form (4.27).*

$$p_{ghr} = \sum_{j \in \mathbb{N}} c_j \prod_{i=1}^n (\Delta^{\mu_{i,j}} - \delta^{\mu_{i,j}})^{\alpha_{i,j}} (\Delta^{\mu_{i,j}} + \delta^{\mu_{i,j}})^{\beta_{i,j}}, \quad (4.27)$$

where each $\mu_{i,j}$ is an arbitrary integer vector.

and each $c_j, \alpha_{i,j}, \beta_{i,j}$ is a non-negative constant.

Proof. $\Rightarrow$: If p_{ghr} is non-negative over the compact set S_c , then it has a Handelman representation; by choosing $\mu_{i,j} = i$, it also has a GHR.

$\Leftarrow$: If p_{ghr} has a GHR, then because any individual variable δ_i is bounded by $|\delta_i| \leq \Delta$, any term can also be bounded over the set S_c , $|\delta^\lambda| \leq \Delta^{|\lambda|}$. This means $\Delta^{|\lambda|} + \delta^\lambda \geq 0$ and $\Delta^{|\lambda|} - \delta^\lambda \geq 0$, and because $c_i \geq 0$, it holds that $p_{ghr} \geq 0$, i.e. non-negative over the set S_c . $\square$

Using this theorem, if we can find GHRs to satisfy (4.28) and (4.29), then the left-hand side is non-negative over the set of inequalities (4.21), and from this a guaranteed bound follows.

$$f(\delta) - \hat{\gamma}_{lower} = p_{lower_ghr}. \quad (4.28)$$

$$\hat{\gamma}_{upper} - f(\delta) = p_{upper_ghr}. \quad (4.29)$$

Our heuristic

Unfortunately, in general it is no easier to find GHRs to satisfy equations (4.28) and (4.29), then it is to find traditional Handelman representations, meaning that an approach using linear programming is not scalable. In this section, we describe our heuristic which is guaranteed to terminate at the same time as aiming to find practically useful bounds.

In our heuristic, we first express $f(\delta)$ in a canonical form, as a sum of monomials in which each term appears at most once. In order to demonstrate this step, consider the earlier example from Table 4.1; the polynomial representation for the variable c in this example can be expanded into the polynomial (4.30) when neglecting the variable δ_3 , since the worst case value of this variable is trivially known to lie at the extremes.

$$\begin{aligned} f(\delta) = & -xy(xy - 1) - xy(2xy - 1)\delta_1 - x^2y^2\delta_2 \\ & -x^2y^2\delta_1^2 - 2x^2y^2\delta_1\delta_2 - x^2y^2\delta_1^2\delta_2 \end{aligned} \quad (4.30)$$

After this expansion, it is possible to ‘cancel’ each individual monomial from the left hand side of equations (4.28) and (4.29), using a polynomial of the form (4.31), and we then note that the sum of

polynomials created in this fashion would be a GHR. After cancelling all the monomials, we would be left with a constant from which the bound $\hat{\gamma}_{lower}$ and $\hat{\gamma}_{upper}$ could be derived in a similar fashion to the earlier example.

$$h(\delta) = c \prod_{j=1}^n (\Delta^{|\mu_j|} - \delta^{\mu_j})^{\alpha_j} (\Delta^{|\mu_j|} + \delta^{\mu_j})^{\beta_j}. \quad (4.31)$$

The complexity of this approach is that there are many monomials in $f(\delta)$ to cancel, and many ways to cancel any given monomial using polynomials of the form (4.31). For example, consider again the polynomial given in (4.30), Table 4.4 illustrates several possible choices of these polynomials that could be used to cancel the highest order monomial $(-x^2y^2\delta_1^2\delta_2)$ from this example. Our proposed heuristic attempts to make the best choices of these polynomials. It is based on the idea that whilst canceling a high order monomial, it is possible to reduce the coefficients of lower order monomials at the same time, as shown in Table 4.4, and this will result in tighter bounds than cancelling each monomial independently. In order to ensure termination, detailed in Section 4.4.2, the heuristic begins by selecting the highest order monomials and chooses polynomials to remove the higher order monomials in such a way that they also reduce the absolute value of the coefficient of the lower order monomials. The overall heuristic is formally given in Figure 4.2; the rest of this section gives a high level discussion of the rationale behind the various stages of the heuristic, while a complete example of our heuristic applied to an instance of (4.30) is given in Appendix A.

Selecting cancellation terms. The first stage involves choosing which lower order monomials are suitable to be modified at the same time as attempting to cancel a higher order monomial. Even for the simple example given in (4.30), the choice of cancellation terms will always depend upon the input. If $2xy > 1$, the first approach from Table 4.4 would be the best of the four approaches as it would decrease the term δ_1 towards zero. On the other hand, if $2xy = 1$ the third approach would be the best of the four as the term δ_1 would already be zero and hence the first two approaches would create a new monomial δ_1 , which is a low order monomial that would have to be later removed. Thus the heuristic searches for non-zero monomials in f whose product will equal the desired higher order term. It performs this search initially looking to build a set using lower order terms as these are the most desirable terms to reduce,

Table 4.4.: Example polynomials of the form (4.27) to cancel the monomial $-x^2y^2\delta_1^2\delta_2$ from f .

Approach Number	1	2	3	4
Handelman Coefficients	$\mu = ([1, 0], [0, 1])$ $\alpha = (0, 0)$ $\beta = (2, 1)$ $c = x^2y^2$	$\mu = ([1, 0], [0, 1])$ $\alpha = (2, 0)$ $\beta = (0, 1)$ $c = x^2y^2$	$\mu = ([2, 0], [0, 1])$ $\alpha = (0, 0)$ $\beta = (1, 1)$ $c = x^2y^2$	$\mu = [2, 1]$ $\alpha = 0$ $\beta = 1$ $c = x^2y^2$
Polynomial h	$x^2y^2(\Delta + \delta_1)^2(\Delta + \delta_2)$	$x^2y^2(\Delta - \delta_1)^2(\Delta + \delta_2)$	$x^2y^2(\Delta^2 + \delta_1^2)(\Delta + \delta_2)$	$x^2y^2(\Delta^3 + \delta_1^2\delta_2)$
New Polynomial $f + h$	$xy(xy - 1 - xy\Delta^3)$ $+xy(2xy - 1 - 2xy\Delta^2)\delta_1$ $+x^2y^2(1 - \Delta^2)\delta_2$ $+x^2y^2(1 - \Delta)\delta_1^2$ $+2x^2y^2(1 - \Delta)\delta_1\delta_2$ $+0\delta_1^2\delta_2$	$xy(xy - 1 - xy\Delta^3)$ $+xy(2xy - 1 + 2xy\Delta^2)\delta_1$ $+x^2y^2(1 - \Delta^2)\delta_2$ $+x^2y^2(1 - \Delta)\delta_1^2$ $+2x^2y^2(1 + \Delta)\delta_1\delta_2$ $+0\delta_1^2\delta_2$	$xy(xy - 1 - xy\Delta^3)$ $+xy(2xy - 1)\delta_1$ $+x^2y^2(1 - \Delta^2)\delta_2$ $+x^2y^2(1 - \Delta)\delta_1^2$ $+2x^2y^2\delta_1\delta_2$ $+0\delta_1^2\delta_2$	$xy(xy - 1 - xy\Delta^3)$ $+xy(2xy - 1)\delta_1$ $+x^2y^2\delta_2$ $+x^2y^2\delta_1^2$ $+2x^2y^2\delta_1\delta_2$ $+0\delta_1^2\delta_2$

and uses higher order terms if necessary.

Selecting a subset of cancellation terms. If we were to create a monomial which reduces all the terms selected in the previous stage, the size of the canonical representation of the form (4.31) grows exponentially in the number of terms. As a result, a tuning factor has been added in the following stage which checks if the number of terms found from the previous stage exceeds a user chosen maximum m . If the number of terms exceeds this maximum, a subset of m terms from the previous stage are chosen, and a single extra high order term is added to ensure the desired monomial still gets cancelled. This tuning factor allows the user to trade execution time for the quality of the bound, and is discussed in more detail in Section 4.6.3.

Choosing signs. Having chosen the terms that are going to be used to create the polynomial, the next stage is to choose the signs. The signs are chosen to reduce as many of the chosen low order monomials as possible, as well as the original highest order term to ensure algorithm termination. For example, from Table 4.4, if $2xy < 1$, the second approach would be the best approach as it would decrease the coefficient of the monomial δ_1 , whereas the first approach would increase this coefficient.

Choosing the initial multiplier. The final stage involves choosing the initial multiplier c for the polynomial h . This is a scalar chosen to be as large as possible whilst ensuring that when the polynomial is added to f , no coefficient in f changes sign. This ensures that at least one coefficient gets cancelled at every iteration of the algorithm.

4.4.2. Algorithm termination

At each iteration of the loop in Figure 4.2, a polynomial h is formed which reduces the absolute value of the coefficient of the highest order monomial in f . The minimum reduction is determined by q , as defined in Figure 4.2, which is a function of terms present in both f and h . After the update, the term from f which determines this minimum reduction value is removed from f . The algorithm then creates a new polynomial h and repeats the process.

Termination is guaranteed because at each iteration of the loop, the absolute value of the coefficient of the highest order monomial in f is reduced by a factor q , which is a function of the absolute values of the coefficients of monomials in f and the cancellation polynomial h , and no higher order monomials

Algorithm $\gamma = \text{BoundPoly}(g, \Delta)$
 // BoundPoly takes a polynomial g in the formal vector variable δ , and a real bound Δ on the
 // absolute value of each element $\delta_i (1 \leq i \leq n)$, returning a lower bound on $g(\delta)$ valid
 // over $\delta \in [-\Delta, +\Delta]^n$.

Set $f = -g$
while f is not constant
 Find greatest monomial in f : $c\delta^\mu$

 // Selecting Cancellation Terms
 Set degree = 0;
 repeat
 degree++;
 Using a greedy search algorithm find a set S where each element is a monomial
 of the form $a_i\delta^{\lambda_i}$ in f such that (letting $n = |S|$):
 $(\sum_{i=1}^n \lambda_i = \mu) \wedge \forall i (|\lambda_i| \leq \text{degree})$
 until (S is nonempty)

 // Selecting a Subset of Cancellation Terms
 if ($n > m$)
 Form a subset $S' \subset S$ of the m lexicographically lowest monomials in S
 Add an extra monomial $c\delta^\mu / \prod_{i=1}^{|S'|} \delta^{\lambda_i}$ to S' to complete the cover
 else
 Let $S' = S$
 Let $n' = |S'|$

 // Choosing Signs
 Let $d\delta^\beta$ be the lexicographically greatest monomial in S' .
 Modify the sign of this monomial by setting:
 $S' = S' \cup \{|d| \text{sgn}(c) \prod_{i=1}^{n'-1} \text{sgn}(a_i) \delta^\beta\} \setminus \{d\delta^\beta\}$.
 for (each monomial $a_i\delta^{\lambda_i}$ in S')
 Create a new polynomial $j_i = (\Delta^{|\lambda_i|} - \text{sgn}(a_i)\delta^{\lambda_i})$

 // Choosing the initial multiplier
 Create $h = \prod_{i=1}^{n'} j_i$
 Let C be the set of terms present both in f and in h .
 For each term δ^{ρ_i} in C , let the corresponding coefficient in f and h be f_i and
 h_i respectively.
 Calculate $q = \min_i (|f_i|/|h_i|)$
 Compute $f = f + qh$

end
 Set $\hat{\gamma} = -f$.

Figure 4.2.: Cancellation algorithm.

are created. If new lower order terms are created, the value of q to remove these whilst cancelling the same high order term will be the same as the value of q which created them, so because q does not decrease, eventually the highest order monomial will be cancelled, ensuring termination.

4.4.3. Division

Because Theorem 3 only applies to polynomials, the above method cannot be directly applied to algorithms including division. However, we note that any computation which consists of the basic operators $\{+, -, *, /\}$ can be converted into a rational function by first applying the multiplicative model of error and then performing simple algebraic manipulation, an example of such manipulation is shown in (4.32); we can then perform an iterative refinement to calculate a bound.

$$\frac{\delta_1}{\delta_2 + \delta_3/\delta_4} = \frac{\delta_1\delta_4}{\delta_2\delta_4 + \delta_3}. \quad (4.32)$$

Assuming the rational representing the range of the chosen output variable is of the form n/d , where n and d are polynomials, then we need to show $n/d \geq \hat{\gamma}_{lower}$ inside the compact set of intent. For $d > 0$, this could be re-written as $n - d\hat{\gamma}_{lower} \geq 0$, which is a polynomial inequality. The previous approach can then attempt to find a GHR for $n - d\hat{\gamma}_{lower}$; if a representation is found for a value of $\hat{\gamma}_{lower}$, we tighten the value of $\hat{\gamma}_{lower}$, if it fails then one can loosen the value of $\hat{\gamma}_{lower}$. A similar approach can be taken for the upper bound.

In order to minimise this search time, this is performed as a binary search with the initial range given by equation (4.33), where the ranges of n and d are found using the original method on the numerator and denominator polynomials alone. Using this range also has the added benefit of checking that the denominator does not include the zero value. Note that if the denominator does contain a zero not cancelled by a zero in the numerator, then no such bound exists in any case.

$$\left[\min\left(\frac{n_{min}}{d_{max}}, \frac{n_{max}}{d_{min}}, \frac{n_{max}}{d_{max}}, \frac{n_{min}}{d_{min}}\right), \max\left(\frac{n_{min}}{d_{max}}, \frac{n_{max}}{d_{min}}, \frac{n_{max}}{d_{max}}, \frac{n_{min}}{d_{min}}\right) \right]. \quad (4.33)$$

4.5. Testing methodology

The motivation for the focus on floating-point hardware is partially due to the relative lack of existing work discussing word-length optimisation for floating-point designs in comparison to fixed-point, and partially due to the recent trends showing that floating-point designs are becoming highly efficient in hardware [146], and the growing collection of publications of floating-point custom hardware implementations [11, 95]. We note, however, that the background theory and heuristic described in this work could easily be adapted to word-length optimisation for fixed-point designs by using a different model of error.

In order to characterise the performance of this work, it has been compared over several tests using a variety of methods. The test cases are those described in the comparative works [83, 84, 158] as well as an iteration of the conjugate gradient algorithm [72] applied to a ‘toy’ matrix of order two (the operations given in Figure 4.3); the interest behind the choice of the conjugate gradient algorithm is to demonstrate our approach can be applied to an example of a real algorithm that is used in finding the solution to a system of linear equations. Though it is acknowledged that for such a small matrix order, the conjugate gradient algorithm is unlikely to be used, it is large in terms of total operations compared to the results reported in [83, 84, 158], and it provides a simple real example on the effects of limited precision computations as well as highlighting the scalability issues. The inputs for the test are shown in Figure 4.3, these are chosen to ensure the matrix is symmetric positive definite, a property required for the convergence of the conjugate gradient algorithm, whilst the input vector is specified as an interval to demonstrate the algorithm can calculate bounds dependent upon input ranges and finite precision. We note that presumably due to run-time issues, the test cases from the comparative works are relatively small and based on the range analysis problem as opposed to the effects of finite precision, but these are included to demonstrate that our approach is also applicable to this problem and performs well in comparison. We then illustrate the greater scalability of our approach relative to these comparative works by demonstrating that it can still find bounds when adding finite precision effects to these problems.

The methods to which our work is compared help to determine where our approach lies within a hierarchy of error bounds, given in Figure 4.4, where the higher the relative error the worse the bound.

$$A = \begin{pmatrix} 10.25 & -9.75 \\ -9.75 & 10.25 \end{pmatrix}, \quad x = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad b \in \begin{pmatrix} [10.25 & 10.75] \\ [9.25 & 9.75] \end{pmatrix}$$

$$\begin{aligned}
d_1 &= b_1 & (1) \\
d_2 &= b_2 & (2) \\
r_1 &= b_1 & (3) \\
r_2 &= b_2 & (4) \\
\delta_{n.t1} &= r_1 * r_1 & (5) \\
\delta_{n.t2} &= r_2 * r_2 & (6) \\
\delta_{new} &= \delta_{n.t1} + \delta_{n.t2} & (7) \\
q_{t1} &= A_{11} * d_1 & (8) \\
q_{t2} &= A_{12} * d_2 & (9) \\
q_{t3} &= A_{21} * d_1 & (10) \\
q_{t4} &= A_{22} * d_2 & (11) \\
q_1 &= q_{t1} + q_{t2} & (12) \\
q_2 &= q_{t3} + q_{t4} & (13) \\
\alpha_{d.t1} &= d_1 * q_1 & (14) \\
\alpha_{d.t2} &= d_2 * q_2 & (15) \\
\alpha_{den} &= \alpha_{d.t1} + \alpha_{d.t2} & (16) \\
\alpha_{cg} &= \delta_{new} / \alpha_{den} & (17) \\
x_{t1} &= \alpha_{cg} * d_1 & (18) \\
x_{t2} &= \alpha_{cg} * d_2 & (19) \\
x_1 &= x_1 + x_{t1} & (20) \\
x_2 &= x_2 + x_{t2} & (21) \\
r_{t1} &= \alpha * q_1 & (22) \\
r_{t2} &= \alpha * q_2 & (23) \\
r_1 &= r_1 - r_{t1} & (24) \\
r_2 &= r_2 - r_{t2} & (25) \\
\delta_{old} &= \delta_{new} & (26) \\
\delta_{n.t1} &= r_1 * r_1 & (27) \\
\delta_{n.t2} &= r_2 * r_2 & (28) \\
\delta_{new} &= \delta_{n.t1} + \delta_{n.t2} & (29) \\
\beta_{cg} &= \delta_{new} / \delta_{old} & (30) \\
d_1 &= \beta_{cg} * d_1 & (31) \\
d_2 &= \beta_{cg} * d_2 & (32)
\end{aligned}$$

Figure 4.3.: Pseudo code for one iteration of the conjugate gradient algorithm on a 2x2 matrix to solve $Ax = b$.

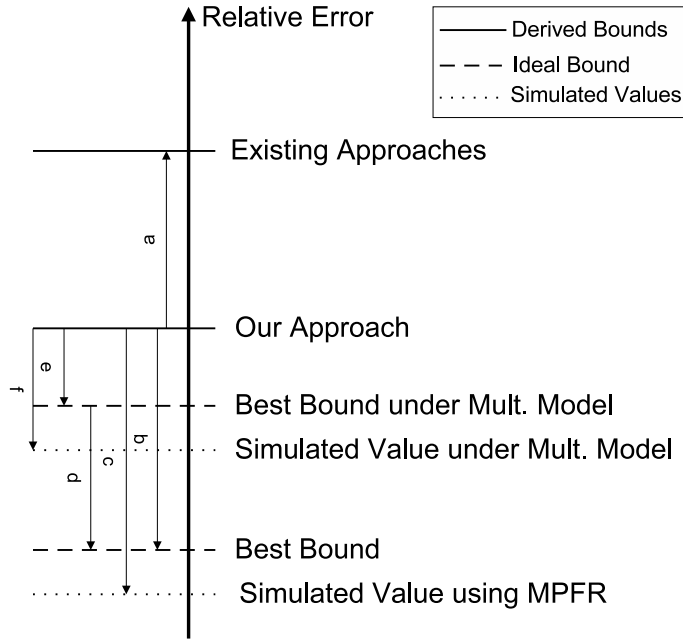


Figure 4.4.: Taxonomy of error bounds.

The initial aim of our results is to demonstrate that our approach is better than existing analytical methods that provide bounds (distance a from Figure 4.4). To this end, we compare our approach against Interval Arithmetic, Affine arithmetic and Taylor models with interval remainder bounds restricted to 1st, 2nd and 3rd order where possible. The reason we are restricted to these small orders is that the Taylor model for inversion is found according to (4.34), where $INT(f(x))$ is the range of $f(x)$ calculated using interval arithmetic, and when a large multivariate polynomial is raised to a high power when calculating $(T_\rho + I_\rho)^i$, the number of monomials grows too large to compute in a reasonable execution time.

$$\frac{1}{(T_\rho + I_\rho)} = \sum_{i=0}^{\rho} (-1)^i (T_\rho + I_\rho)^i + \frac{(-1)^{\rho+1} INT((T_\rho + I_\rho))^{\rho+1}}{INT((T_\rho + I_\rho))^{\rho+2}}. \quad (4.34)$$

It is also desirable to find out how far our approach lies from the ideal bound (as shown by distance b). Unfortunately, finding the ideal bound is NP-hard, for it would involve calculating the potential values of every variable and from these values determining the worst case error loss for every precision. Instead, this is approximated by Monte Carlo simulation using the MPFR multiple precision floating library [56]

(distance c). However, one must note that such an approach no longer returns a provable bound, and the difference between our approach and this bound is a function of three factors: the quality of our calculated bound, the simulation time, and the accuracy of the standard multiplicative model of floating-point error. The model of error used in this work, and throughout the numerical analysis literature, is in practice a conservative approximation because each δ_i is a function of the input variables, but this information is lost; as an example, multiplying any value by any power of two will be error free in the absence of overflow or underflow. Therefore there will exist a distance d between the ideal bound and the best any approach using this model of error could possibly achieve. As such, we are most interested to find out how far our approach differs from the best possible bound achievable using the model of floating-point error generally used in numerical analysis literature (distance e). Unfortunately, as has been discussed in Section 4.3, finding the best bound under the multiplicative model of floating-point error involves polynomial optimisation, which is NP-hard. Therefore, this too is approximated under our test labelled ‘Random sampling over poly’, where we apply Monte Carlo simulation over the relevant ranges for all the variables to the polynomial we created which bounds the range or relative error of the result (distance f). It should be noted that if distance f can be shown to be small, this will imply our approach lies close to the optimal bound achievable e (since it is known *a priori* that $f \geq e \geq 0$). Finally, distance c is also reported, so as to gain an approximate estimate of the appropriateness of the multiplicative model of error. As both c and f are simulated values, it cannot be guaranteed that $c \geq f$, but if c is much larger than f , it is likely to be a result of the conservative nature of the multiplicative model.

As our approach consists of several stages, the main stages of which are creating a simplified polynomial and finding a bound for this polynomial using Handelman representations, to help quantify the contribution of each of these stages, we compare the result of bounding the polynomial in canonical form by applying interval arithmetic against our full approach. One should note that for algorithms which do not include division, the test of bounding the simplified polynomial using interval arithmetic is comparable to applying Taylor models approach without any intermediate bounding of higher order terms, which would give the tightest bound achievable using Taylor models. For algorithms which do involve division, our test which bounds the simplified polynomial using interval arithmetic simply bounds the

numerator and denominator polynomial separately and performs interval arithmetic on these two results instead of using the approximation given by (4.34). This however remains an interesting test as it allows us to focus on quantifying how our method of applying Handelman representations to bound a rational function can find tighter bounds than any approach that is fundamentally based upon interval arithmetic.

4.6. Results

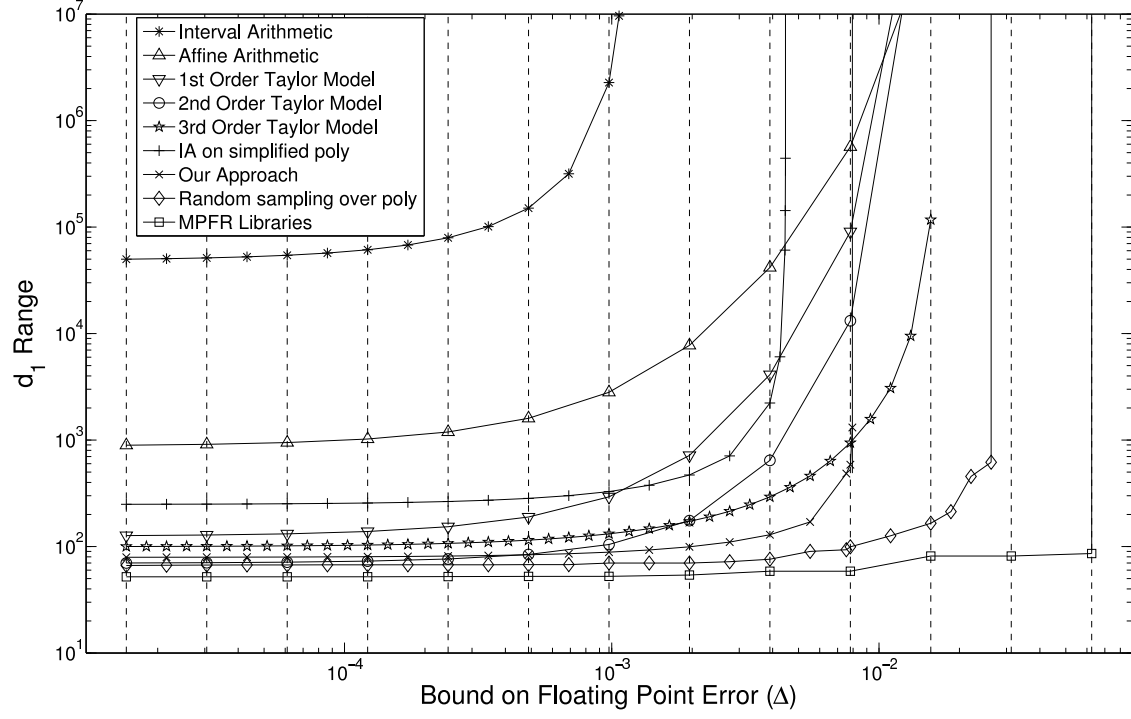
4.6.1. Analysis of our approach on an iteration of the 2x2 conjugate gradient algorithm

The effects of finite precision on range

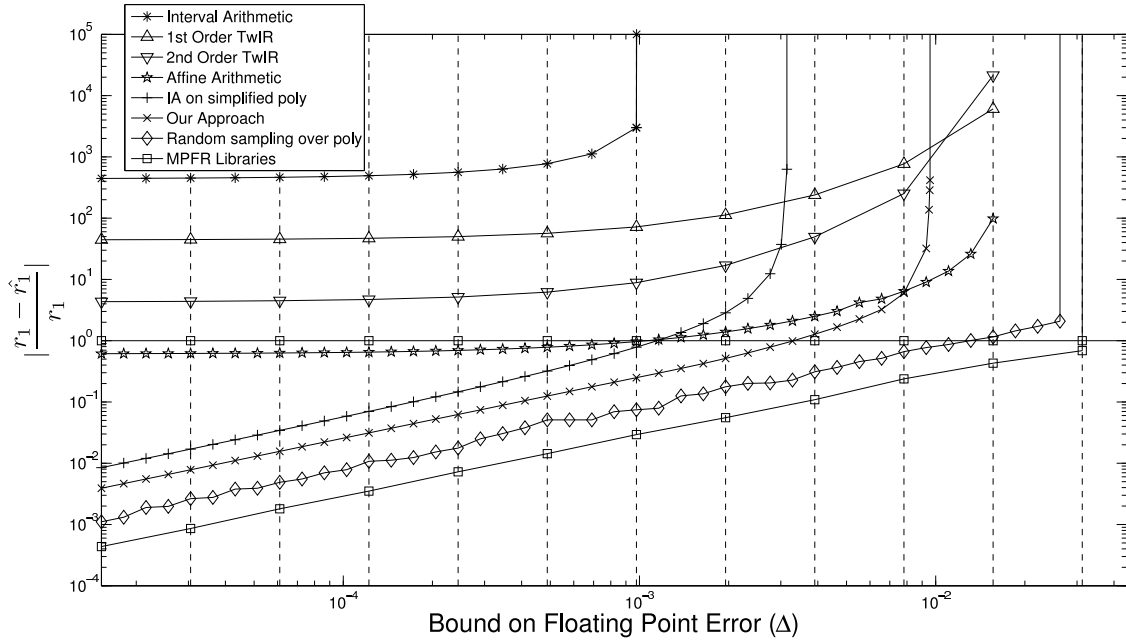
Figure 4.5(a) highlights the effect precision has on the conjugate gradient algorithm and the quality with which our approach can characterise this effect. It shows how the bounds on the range of the variable d_1 after an iteration of the conjugate gradient algorithm (operation 31 from Figure 4.3) changes as a function of the precision, for the tests mentioned in Section 4.5. On these graphs, the vertical dotted lines illustrate the values of precision for realisable word-lengths, *i.e.* the difference in word-length between any two adjacent dotted lines is one bit. It is clear to see there is a significant difference between the ranges calculated by all approaches in comparison to interval arithmetic. In comparison to other approaches, our approach generally finds the tightest bounds, the exception being when the precision is very small where affine arithmetic and Taylor series methods can calculate bounds where our approach fails. The reason for this is in these cases, bounds for the error variables are proportional to the bounds on the input ranges, so the first order approximation of division retains most of the information, whereas our heuristic struggles without such a simplification.

It is also interesting to note that this graph demonstrates that both stages of our approach provide significant benefits towards obtaining a better bound, for performing interval arithmetic on the simplified polynomial is significantly better than interval arithmetic in the traditional sense, whilst bounding the simplified polynomial using Handelman representations improves this bound even further.

In order to view how these results translate to actual hardware savings on a FPGA, Table 4.5 shows the number of slices required, the latency, and the maximum frequency achievable when using single or double precision units, or by performing optimisation for a given bound on range using all approaches



(a) Bounds on the range of the variable d_1 (operation 31 from Figure 2.1).



(b) Bound on relative error of conjugate gradient 'Residual'. r_1 (operation 24 from Figure 4.3) is the nominal 'residual', whilst $\hat{r}_1$ is the residual taking into account floating-point error.

Figure 4.5.: Range and relative error results for various operations in conjugate gradient example.

Table 4.5.: Resource usage, max frequency and latency of conjugate gradient implementations.

Method	Precision (# bits)	Slices	Frequency (MHz)	Latency (cycles)
Our Approach	7	1663	367	161
Affine Arithmetic	8	1710	360	165
3rd Order Taylor	9	1923	350	169
IA on Simplified Poly	9	1923	350	169
2nd Order Taylor	10	1964	314	173
IEEE Single Precision	23	5587	286	277
IEEE Double Precision	52	15672	143	425
1st Order Taylor	∞	∞	N/A	N/A
IA	∞	∞	N/A	N/A

Table 4.6.: Comparison of execution times to compute range of d_1 and relative error of r_1 for a given precision.

Method	Average time to compute range of d_1 (s)	Average time to compute relative error of r_1 (s)
Interval Arithmetic	0.003	0.003
Affine Arithmetic	3.7	3.5
Taylor Model of 1st Order	5.3	8.26
Taylor Model of 2nd Order	48	53
Taylor Model of 3rd Order	3700	N/A
Our approach (set-up time)	3660	150
Our approach (each iterative refinement)	2100	35

targeting a desired range of 600. In this test, these figures are post place and route results where the floating point components are generated using Xilinx Coregen [157]. From this table, our approach can achieve a reduction in slices in comparison to optimising the design using the other approaches, and significantly larger savings in comparison to using full IEEE single or double precision arithmetic. It also demonstrates the design would run at a faster frequency and an iteration would complete in fewer cycles by performing the optimisation.

The effect of finite precision on relative error

Figure 4.5(b) demonstrates the bound on relative error. This figure clearly demonstrates that the relative error decreases with precision, and that our algorithm is capable of tracking this relationship well, unlike

all the other methods operating on the original code. The reason the other approaches cannot track this relationship is that almost all of the relative error terms will be of second order or greater because they will be a function of the input variables multiplied by some finite precision error. However interval arithmetic retains no information about the polynomial, whilst affine arithmetic only retains first order information and approximates higher orders, and the tests using Taylor models only retain first and second order information. As a result, the bounds reported will be based on approximations of higher order terms, and these approximations must be treated independently, therefore all dependencies are lost. In contrast, by retaining all the information throughout forming the polynomial, as shown by applying interval arithmetic on the simplified polynomial, it is possible to obtain better bounds, whilst the benefit of our approach in handling dependencies within the polynomial using Handelman representations results in even tighter bounds.

Finite precision effects throughout an algorithm

Figure 4.6 demonstrates how the error as a result of using a chosen finite precision grows throughout the conjugate gradient algorithm. The abscissa represents the operation in the algorithm, corresponding to the line numbers given in Figure 4.3. This demonstrates that as the number of dependencies grow, the relative error grows, along with the difficulty to bound this error. This graph also highlights some of the deficiencies of simulation methods: at several points, the relative error is high for both the simulation methods, but undefined using the two analytical approaches. Upon further inspection, it can be shown that over the specified input range, the denominator in the relative error term ($n\hat{d}$) can legitimately include zero, as a result of input ranges. This demonstrates the limitation of the simulation approach.

Execution time of our approach on conjugate gradient

It should however be mentioned that analysing bounds on the range of the variable d_1 or d_2 (lines 31 or 32 of Figure 4.3) and the relative error for the value r_1 or r_2 (lines 24 or 25 of Figure 4.3) after one iteration was as far through the conjugate gradient algorithm that our approach could calculate bounds in a reasonable amount of time. The times for all approaches to calculate the polynomial, calculated as an average over many test runs on an Intel Xeon E5345, are given in Table 4.6. In this table, because the

conjugate gradient algorithm includes division, our approach requires the iterative refinement mentioned in Section 4.4.3, so we have separated the computation time for our approach into two stages: set-up time - the time to calculate the initial bounds to perform the iterative refinement, and the time for each iterative refinement. As many iterative refinements are required to calculate the bounds for each point on the graph, this is as far through the algorithm that we could calculate in a reasonable time, especially given the scalability issues which will be discussed in Section 4.6.4. In comparison to interval arithmetic and affine arithmetic our approach is significantly longer as these approaches significantly limit the number of monomials in the polynomial, allowing a faster solution, where in contrast, when finding the range of d_1 , our heuristic was applied to a polynomial consisting of approximately 2 million monomials. The execution time of our approach is more comparable to Taylor models of higher orders firstly because they retain many more monomials and secondly because the time to compute the approximation for division is large.

While the execution time for computing the relative error in Table 4.6 appears much smaller, we cannot bound a variable further through the conjugate gradient algorithm because the function representing relative error is much larger in degree and number of monomials than the associated polynomial representing range. To explain this, let us describe the nominal value of the rational function by n/d , and the rational function including floating-point error by $\hat{n}/\hat{d}$. Using this notation, for the graph in Figure 4.5(a), we find the bounds of the rational function $\hat{n}/\hat{d}$ for the upper and lower bounds, whereas when finding the relative error in Figure 4.5(b), we find bounds of the rational function $[(\hat{n}/\hat{d}) - (n/d)]/(n/d)$ or $(\hat{n}d - \hat{d}n)/(n\hat{d})$. With this larger function, the squaring operation (operation 27 in Figure 4.3) results in a polynomial that is too large to compute in tractable time. This additional complexity also affects other approaches, notably the computation of $1/(\hat{r}_1)$ takes too long and too much memory to compute using a 3rd order Taylor approximation. Note, however, that the size of this benchmark is still significantly greater than those reported in [83–85].

4.6.2. Range analysis vs other works

Figures 4.7(a) and 4.7(b) show the performance relative to the approaches given in related works, with the actual values given in Tables 4.7 and 4.8. In Figure 4.7(a), the ranges for the various approaches

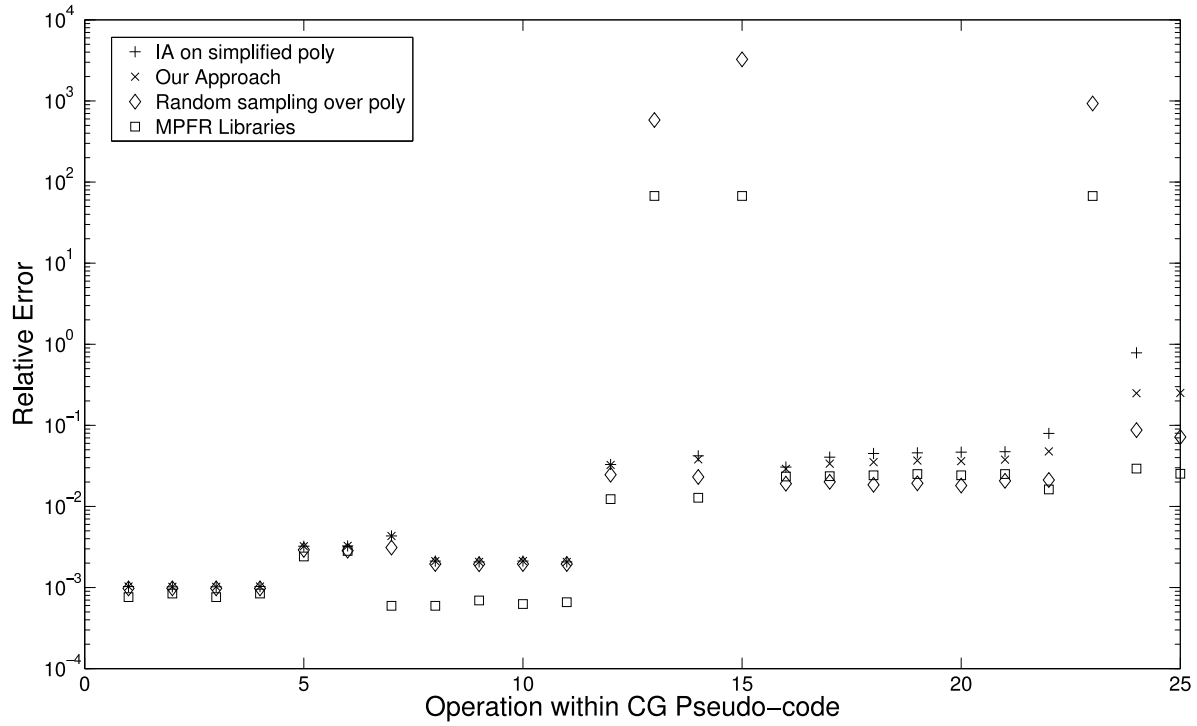
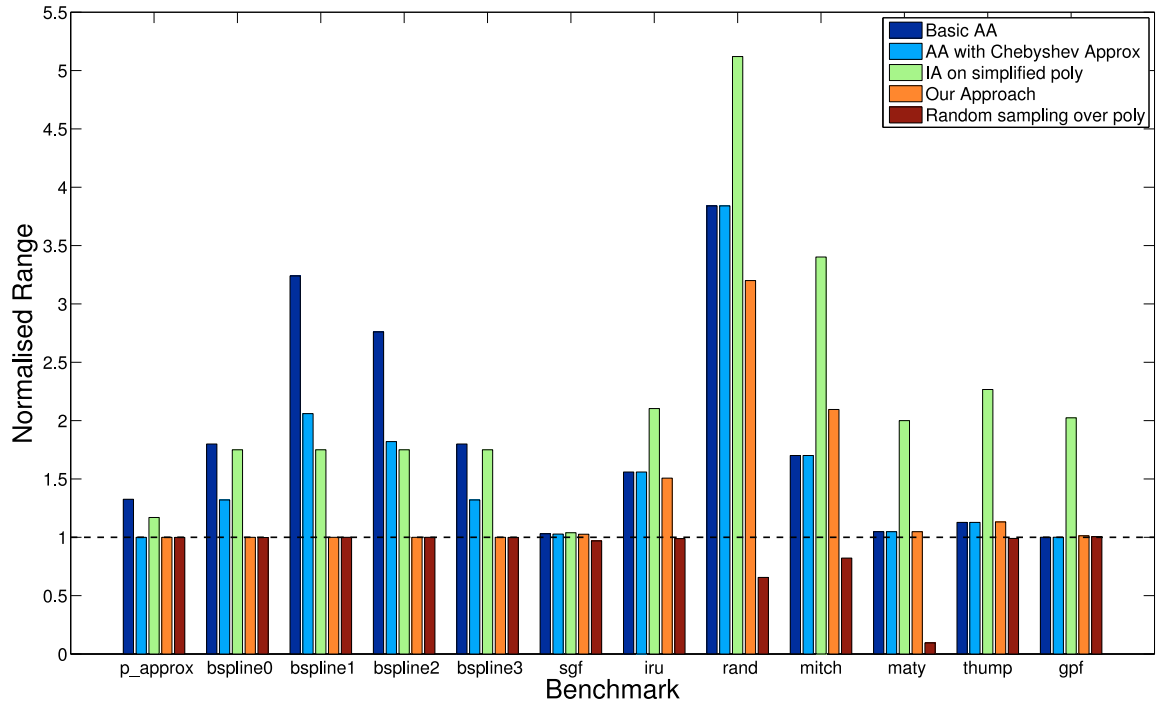
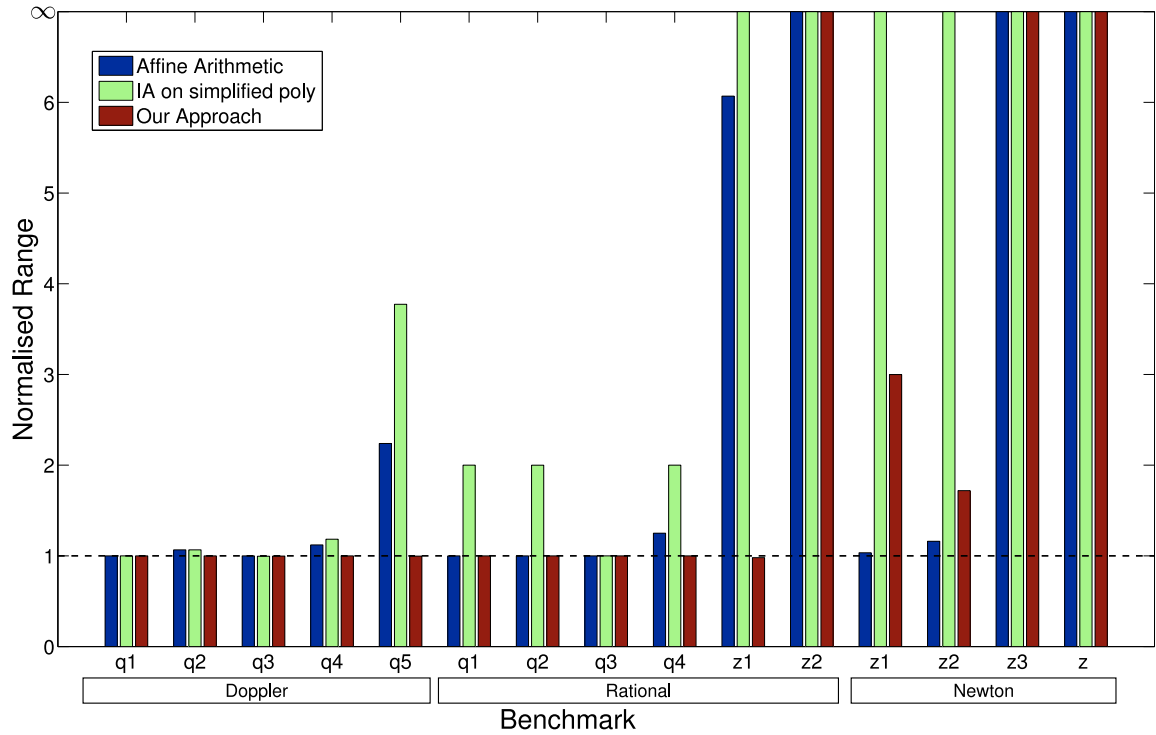


Figure 4.6.: Growth in relative error throughout a conjugate gradient iteration. Operations correspond to pseudo code in Figure 4.3.



(a) Comparison vs AA. Range widths found for the benchmarks using our approach and AA are normalised with respect to the ‘ideal’ values stated in [158].



(b) Comparison vs SMT. Range widths found for the benchmarks using our approach and AA are normalised with respect to the values stated in [84, 85].

Figure 4.7.: Range comparison against published methods.

Table 4.7.: Comparison of our approach vs AA using Chebyshev approximations approach [158].

	Infinite Precision											
	Basic AA		AA with Chebyshev Approx		SIA		Our		Monte Carlo		Ideal	
	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper
Poly Approx	-0.0541	0.865	0	0.6931	0	0.8108	0	0.6932	0.0003	0.6929	0	0.6931
B - Spline 0	-0.13	0.17	-0.05	0.17	-0.125	0.1667	0	0.1667	0	0.1664	0	0.1667
B - Spline 1	-0.33	1.29	-0.05	0.98	0.0417	0.9167	0.1667	0.6667	0.1667	0.6667	0.1667	0.6667
B - Spline 2	-0.21	1.17	-0.02	0.89	0.0417	0.9167	0.1667	0.6667	0.1667	0.6667	0.1667	0.6667
B - Spline 3	-0.17	0.13	-0.17	0.05	-0.1667	0.125	-0.1667	0	-0.1666	0	-0.1667	0
sgf	-9803	9525	-9793	9487	-9821	9671	-9765	9487	-9301	8874	-9453	9303
iru	-95000	128000	-95000	128000	-148350	152450	-91390	124160	-53581	87743	-55100	87900
rand	-192	192	-192	192	-256	256	-192	128	-29.2096	36.432	-36	64
mitch	-223	881	-223	881	-1087	1121	-719	641	-7.9817	525.5058	-8	641
maty	-4800	100000	-4800	100000	-100000	100000	-4800	100000	0.2	9487.4	0	100000
thump	-60000	1000000	-60000	1000000	-1065400	1065400	-62400	1001200	0	930990	0	940000
gpf	-1.2E+08	1.19E+08	-1.2E+08	1.13E+08	-8416264	8417464	-7261200	7098000	0	1013500	3	957000
rat	-2.1E+08	3.3E+11	-2.1E+08	3.3E+11	-3.3E+11	3.34E+11	-6.1E+08	3.34E+11	0	3.32E+11	-1.03	3.3E+11
With Finite Precision ($\Delta = 2^{-10}$)												
	Our		SIA		Monte Carlo							
	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper
Poly Approx	-0.00331	0.814183	-1E-04	0.696197	0.000556	0.694708						
B - Spline 0	-0.12598	0.167646	-2.8E-17	0.167646	3.8E-11	0.167066						
B - Spline 1	0.036691	0.921643	0.16325	0.668954	0.166967	0.667602						
B - Spline 2	0.036689	0.921644	0.166341	0.671402	0.16681	0.668224						
B - Spline 3	-0.16683	0.125163	-0.16683	1.39E-17	-0.16645	-4.7E-15						
sgf	-9930.96	9780.957	-9949.73	9608.532	-9245.45	8817.592						
iru	-149425	153520.6	-91996.1	124927.5	-54465.3	87326.63						
rand	-258.259	258.2588	-193.41	242.08	-33.5023	32.21554						
mitch	-1095.63	1129.625	-716.476	664.7615	-7.97463	538.1909						
maty	-10034.4	10034.42	-4814.08	10034.42	0.298266	9870.291						
thump	-1072698	1072698	-63759	1069686	2.09488	937919.5						
gpf	N/A	N/A	N/A	N/A	N/A	N/A						
rat	-3.4E+11	3.37E+11	-9.4E+08	3.37E+11	2234.393	3.31E+11						

Table 4.8.: Comparison of our approach vs SAT modulo theory approach [83, 84].

	Infinite Precision							
	AA		SIA		Our		SAT Mod	
	Lower	Upper	Lower	Upper	Lower	Upper	Lower	Upper
Doppler q1	313	362	313	362	313	362	313	362
Doppler q2	-473252	7228000	-473252	7228000	6268	7228000	6267	7228000
Doppler q3	213	462	213.4	461.4	213	461.4	213	462
Doppler q4	25363	212890	14790	212890	45539	212890	45539	212890
Doppler q5	-80	229	-32.0034	488.7892	0.0339	137.6386	0	138
Rational q1	125	250125	-249875	250125	125	250125	124	250126
Rational q2	1	10001	-9999	10001	1	10001	0	10002
Rational q3	-20000	20000	-20000	20000	-20000	20000	-20001	20001
Rational q4	-2.5E+07	1E+08	-1E+08	1E+08	1	1E+08	0	1E+08
Rational z1	-250	369	FAIL	FAIL	25.01	125	24	126
Rational z2	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	-67	67
Newton z1	-1250360	1170360	FAIL	FAIL	-3615080	3405080	-1205361	1135361
Newton z2	-5753	35769	FAIL	FAIL	-2.57E+04	3.58E+04	1	35769
Newton z3	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	-39	38
Newton z	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	-69	72
	With Finite Precision ($\Delta = 2^{-10}$)							
			SIA		Our		Monte Carlo	
			Lower	Upper	Lower	Upper	Lower	Upper
Doppler q1			313.0177	361.7823	313.0764	361.7823	313.1927	361.5274
Doppler q2			-487963	7242711	5903.514	7242711	7154.64	7183615
Doppler q3			212.5668	462.2332	212.8682	462.2332	218.0062	460.9534
Doppler q4			13809.32	213868.2	45261.52	213868.2	48330.72	208073.4
Doppler q5			-35.848	524.992	0.032124	138.6192	0.088272	127.0179
Rational q1			-250608	250858.3	124.8779	250858.3	125.0591	250134.7
Rational q2			-10018.5	10020.54	0.999023	10020.54	1.00445	9999.922
Rational q3			-20019.5	20019.53	-20019.5	20019.53	-19968.4	19980.35
Rational q4			-1E+08	1.01E+08	0.997073	1.01E+08	1.000176	99690820
Rational z1			-25.0639	25.05885	24.91244	126.8	24.94171	124.9827
Rational z2			FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
Newton z1			-2487036	2507516	-2487057	2227621	-2395376	2164654
Newton z2			-292243	292307.4	-200614	292307.4	-139.109	275900.8
Newton z3			FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
Newton z			FAIL	FAIL	FAIL	FAIL	FAIL	FAIL

are normalised against what is quoted as the ideal range given in [158]. It is interesting to note that in many of the test cases our approach matches the ideal, and in all but one our approach is superior to the other methods. As these examples do not include any finite precision effects, this difference is not a function of the model of error, rather the fact that our approach does not perform well in this one case demonstrates that our heuristic does not always find the best Handelman representation. This is likely to be a result of targeting our heuristic mainly towards minimising finite precision errors as opposed to range analysis. Once again, this graph also illustrates that both stages in our algorithm have significant effects on the final quality of the bound, since simply applying the first stage to create the polynomial and using interval arithmetic on this polynomial can give comparable results to the existing methods for many of the benchmarks, whilst applying the second stage improves the bound further. It is also important to note at this stage that all the test cases in Figure 4.7(a) and some in Figure 4.7(b) are only for polynomials, and hence the results for interval arithmetic on the simplified polynomial are equivalent to the best one could achieve using Taylor models without any intermediate bounding of higher order terms, and this demonstrates how our approach outperforms this method. Finally, the random sampling of values of δ on the simplified polynomial is included to show that the difference between simulation and the best possible bound (distance (c-b) from Figure 4.4) exists even for these simple examples consisting of only three variables. This helps to emphasise that the bounds found in the conjugate gradient analysis are very accurate, given that they consist of up to 25 variables for the polynomial minimised in Figure 4.5(a).

In Figure 4.7(b), the various approaches are normalised against the ranges given in [83, 84], which given sufficient run time, should be optimal. Interestingly, our approach in some cases gives slightly superior results. This is likely to be a result of the fact the SMT is a refinement process which is potentially time consuming and hence the refinement stops once at a given level of accuracy. On the other hand, it is important to note that in some cases SMT does outperform our approach to the extent that in some cases our work gives unbounded results, whereas SMT solver returns bounds. Furthermore, our approach either outperforms or is equivalent to affine arithmetic in all but two of these benchmarks. Similarly to the previous example, this is a limitation of the method at finding the best Handelman representation, and whilst achieving better bounds is possible, it would require a more sophisticated

Table 4.9.: Execution time of our approach and number of monomials for AA benchmarks [158].

	Infinite Precision		
	Time using AA (ms)	Time using Our Approach (ms)	Number of Monomials
Poly Approx		232	20
B - Spline 0	85.8	176	4
B - Spline 1	94.0	143	4
B - Spline 2	95.1	149	4
B - Spline 3	83.5	173	4
sgf	1288.9	370	10
iru	1327.2.9	186	11
rand	413.9	304	9
mitch	764.8	209	10
maty	288.4	124	3
thump	627.3	287	5
gpf	2545.8	300	45
rat	1053.2	227	6
With Finite Precision ($\Delta = 2^{-10}$)			
	Time using AA (ms)	Time using Our Approach (ms)	Number of Monomials
	N/A	2588	1280
	N/A	214	8
	N/A	722	624
	N/A	525	304
	N/A	466	128
	N/A	184527	11738
	N/A	6378	4440
	N/A	22133	4608
	N/A	9379	2786
	N/A	285	40
	N/A	551	224
	N/A	N/A	N/A
	N/A	4767	1080

Table 4.10.: Execution time of our approach and number of monomials for SMT Benchmarks [83].

Infinite Precision		
	Time using Our Approach (ms)	Number of Monomials
Doppler q1	118	2
Doppler q2	148	4
Doppler q3	116	3
Doppler q4	176	6
Doppler q5	3000	10
Rational q1	121	2
Rational q2	124	2
Rational q3	114	1
Rational q4	189	3
Rational z1	4000	4
Rational z2	N/A	N/A
Newton z1	171	7
Newton z2	142	6
Newton z3	N/A	N/A
Newton z	N/A	N/A
With Finite Precision ($\Delta = 2^{-10}$)		
	Time using Our Approach (ms)	Number of Monomials
	165	8
	292	32
	203	18
	576	216
	31000	280
	238	10
	189	6
	145	2
	314	36
	13000	26
	N/A	N/A
	691	29
	517	8
	N/A	N/A
	N/A	N/A

¹ Exact times not reported, but all benchmark times of order 100s in [83,84].

search for Handelman representations, which will be time consuming. However, it should be mentioned that our solver calculates these values within a hundreds of milliseconds, as shown in Tables 4.9 and 4.10, whereas in [84], the results for each value are reported as around the order of 100 seconds on a comparable machine. Furthermore, one should also note that our approach could be used as an input to the SMT solver, which uses initial estimates based on interval or affine arithmetic, to decrease the time to find a good solution.

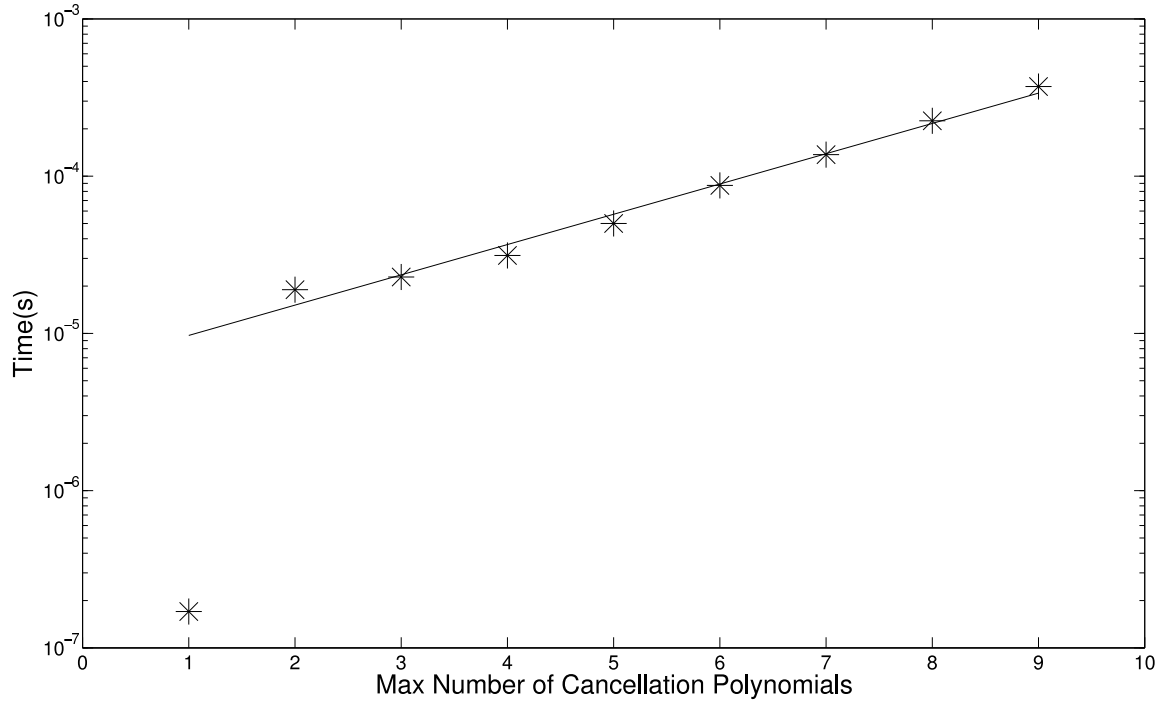
Benchmark tests in finite precision

Tables 4.7 and 4.8 give a list of various results over the benchmarks mentioned earlier, along with the addition of finite precision effects. This is included both to provide data for future works to compare to, as well as to demonstrate how finite precision can affect such bounds. It also shows the benefit of this approach over those given in the comparative works seeing as ours is far more scalable as it can find a solution over many more variables in tractable time: most values are still calculated within hundreds of ms (on a standard desktop with an Intel Core2Duo E6850 processor), as seen in Tables 4.9 and 4.10. Though there are some values that take significantly longer (up to half a minute), these are due to the iterative refinement to compute bounds for division, with the number of iterations of the algorithm proportional to the desired precision of the bound.

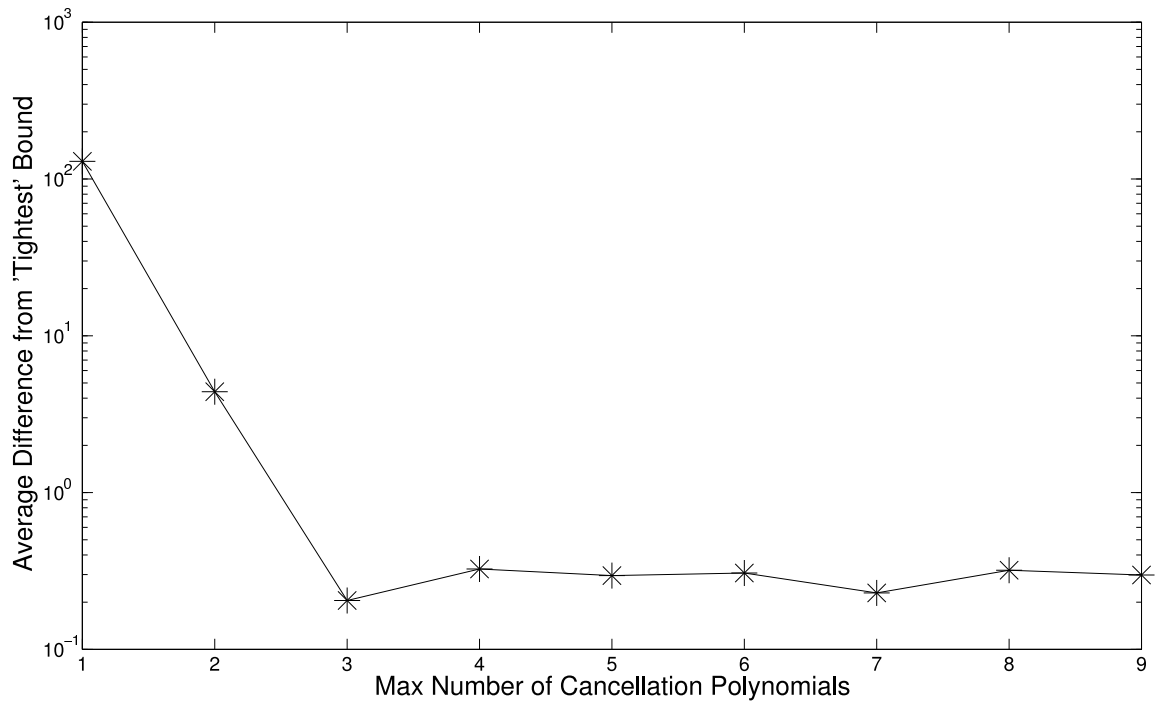
4.6.3. Choosing the maximum number of cancellation terms

The choice of the number of cancellation terms, the value of the variable ‘ m ’ in Figure 4.2, has been chosen experimentally. Figures 4.8(a) and 4.8(b) show how the execution time and error respectively change with m . This section discusses how they were used to select a value of ‘ m ’.

The execution times plotted in Figure 4.8(a) are the average time to compute the variables j , h , q and f from Figure 4.2, as the rest of the execution time should be independent of the variable m . The growth in execution time, as can be seen in Figure 4.8(a), is approximately exponential in m . This is expected given that the number of monomials in $h(= \prod_{i=1}^{n'} j_i)$ is worst case exponential in m , and this means the number of computations to choose the value q and perform the update of f will also grow exponentially. The deviations from the line of best fit are likely to be caused by the fact that $n' \leq m$, as opposed to



(a) Average Execution Time to Compute update of 'm' Cancellation Polynomials.



(b) Average Distance From 'Tightest' Calculated Bound for Product of 'm' Cancellation Polynomials.

Figure 4.8.: Graphs illustrating the effect of changing the number of cancellation polynomials ('m').

exactly equal to m , changing the number of computations for some iterations. The computation time for $m = 1$ is so low because this is a special case in that it is equivalent to performing interval arithmetic on the expanded polynomial, which is significantly faster. Finally, one should note that the choice of m can also impact the number of iterations of the algorithm, but this is hard to quantify for practical examples as it is highly dependent upon the input polynomial.

Because it is impractical to calculate ideal bounds, in order to gain an insight into the quality of the GHR as m increases, the algorithm in Figure 4.2 is applied to a polynomial with various values of m and the difference is calculated between the bound for each value of m and the tightest bound out of all the tested values of m returned for that polynomial. This process is then repeated over several polynomials to obtain an average, which is plotted in Figure 4.8(b). From this figure, it is clear that when $m = 1$, the result is significantly worse, demonstrating again the value of applying our procedure as opposed to performing interval arithmetic on the polynomial. However, for $m \geq 3$, the quality of the best bound is unclear, as seen in Figure 4.8(b). This demonstrates the intricacies involved in choosing the best GHRs. Our algorithm was based upon the idea of reducing the coefficients of lower order monomials at the same time as cancelling higher order monomials. When m is larger, it will create more monomials and these can potentially reduce more lower order monomials, but whilst the algorithm is designed to attempt to choose these terms to reduce the coefficients of the monomials in f , it is not guaranteed, indeed it is even possible to create some unwanted higher order monomials that are products of the lower order monomials. However, more importantly, when m is large, many of these terms created in h will be very small as they will be multiplied by Δ raised to high powers, meaning that for $m > 3$, it is expected that there will be little quantifiable gain in quality of result, and since the execution time grows exponentially with m , the choice of $m = 3$ would appear to be ideal in practice.

4.6.4. Scalability of our approach

Figure 4.9 demonstrates how the execution time of our heuristic to bound the polynomial grows with the number of monomials in the polynomial g . It is clear from this graph that execution time grows at a steady rate with the number of monomials. The rate of growth is slightly super-linear, which is a result of typically creating a GHR for each term and the complexity of creating a GHR increasing

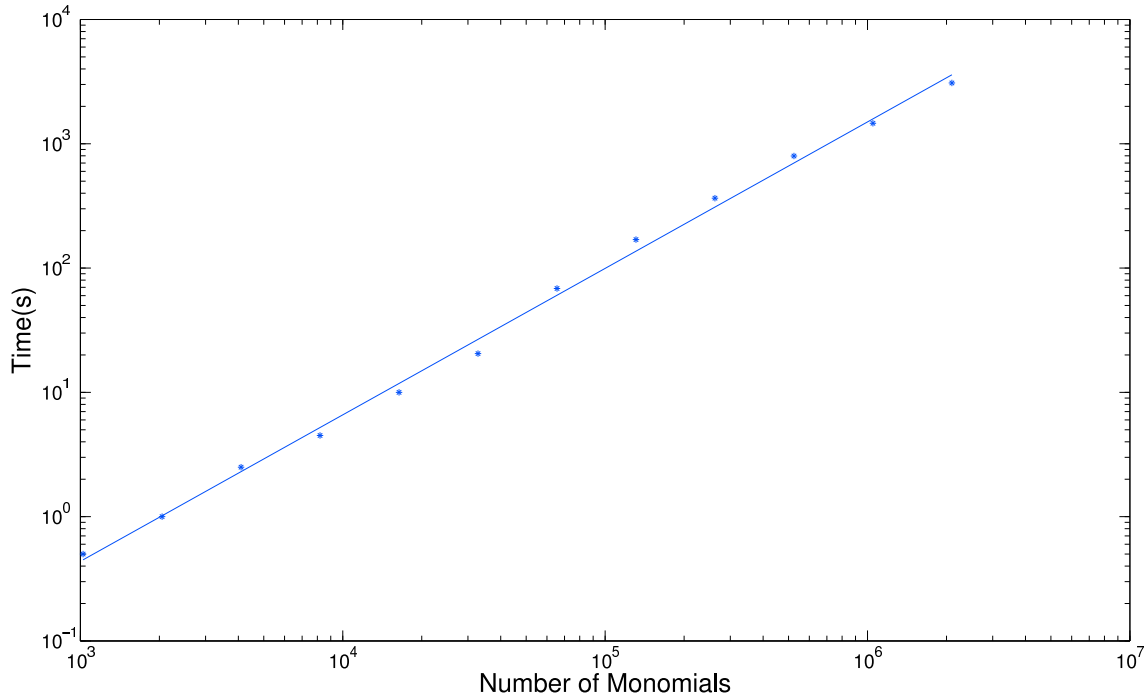


Figure 4.9.: Execution Time growth with number of monomials.

for higher degree monomials and the polynomials with more monomials containing more higher degree monomials. We note that this is much more favourable than the approach using linear programming, as mentioned in Section 4.3.

However, whilst the execution time is only slightly super-linear in the number of monomials used for the canonical polynomial representation, the number of monomials grows significantly faster. This is because in order to correctly bound the error introduced by the use of floating point precision, the polynomial representing the result of any operation is multiplied by the function $(1 + \delta_i)$, according to the multiplicative model of error used throughout the numerical analysis literature, doubling the number of monomials in the representation. For example, the floating-point error model for an algorithm consisting of a series of floating-point multiplications, an example pseudo code for which is shown in Figure 4.10, will have a polynomial representation as in (4.35); the number of monomials in this polynomial, when expanded into canonical form, is exponential in n . Furthermore, the size of the polynomial could grow even faster if each x_i were themselves polynomials.

```

Algorithm  $y = VectorProduct(x)$ 
y=1;
for i=1 to n
    y = y*x[i];
end

```

Figure 4.10.: Pseudo code to calculate the product of vector elements.

$$x_1 x_2 \dots x_n (1 + \delta_1)(1 + \delta_2) \dots (1 + \delta_n) \quad (4.35)$$

While this worst case does not occur in many algorithms, firstly because results are often not accumulated using a single variable throughout the course of the algorithm, and secondly because it is highly likely that a large amount of cancellation will occur, the scalability of the algorithm remains a major issue which will be addressed in chapter 5.

4.7. Summary

This chapter has demonstrated a heuristic, based upon a result from real algebraic geometry, that can be used to find analytical bounds for any value within an algorithm. This approach offers a new alternative to calculate bounds for a polynomial which has no reliance on interval arithmetic, unlike affine arithmetic (when bounding non-affine terms), Taylor series with interval remainder bounds (when bounding the final polynomial as well as high order terms and any actions with the remainder bounds), and even the inner loop of SMT (inside the HySAT solver) which are all dependent upon interval arithmetic to some degree. The main advantage of not relying on interval arithmetic is that our method can consider some of the dependency information within any polynomial in order to calculate tighter bounds. Furthermore, any combined method as bounding using Handelman representations and some sort of Taylor approximations will still be capable of handling dependencies within any resultant polynomial, meaning it would be able to calculate tighter bounds than a direct implementation of these methods.

Whilst this research has shown a high degree of utility, there are several limitations that would benefit

from further research. These include scalability, highlighted in Section 4.6.4, improving the heuristic to obtain better Handelman representations, discussed in Section 4.6.2, as well as extending the method to handle non-polynomial functions such as square roots and exponentials. We address these issues to some degree in the following chapter, whilst noting that the Handelman representation is just one special case of ‘theorems of alternatives’ in which real algebraic geometry is rich. In particular, the proposed approach can be seen as a search over a particular form of Positivstellensatz refutation [126], an insight which could lead to further sophisticated algorithms developed in this field.

Chapter 5

A scalable precision analysis technique

In the previous chapter, we described a new approach to calculate bounds on the range or relative error of a variable in an algorithm. We demonstrated how this approach is capable of reducing the effect of widening of bounds due to dependencies in a multivariate polynomial, resulting in tighter bounds to those achievable by traditional methods including interval arithmetic, affine arithmetic and Taylor series methods with interval remainder bounds. It showed further advantages in the case of division, in that by using a rational expression instead of a polynomial, it retains any correlation between the numerator and denominator, unlike Taylor forms which must approximate the reciprocal function using a polynomial, resulting in a loss of this information, an issue we highlighted in Chapter 2. It also showed that on flexible hardware, these bounds can be translated into improvements in terms of silicon area, operating, frequency and latency when creating a custom accelerator with guaranteed numerical properties. Unfortunately, in the previous chapter, we also highlighted that this approach was limited to small problems because it made no effort to control the size of the polynomial, and was also only applicable to problems consisting of the basic algebraic operators ($\odot \in \{+, -, *, /\}$).

In this chapter, we attempt to address these issues and design an approach which can scale to larger examples that consist of any elementary functions, and still obtain tighter bounds for any given algorithm over a set of input ranges and precision than the existing analytical methods. Furthermore, the approach we discuss in this chapter gives us a degree of control over the trade-off between quality of bounds and execution time. This differs from the existing analytical techniques which have an execution time that is a function of the number of input variables and the code size, as we discuss in greater detail in Section 5.1.

Flexibility over execution time may be of interest within an optimisation framework. For example, an optimisation tool may wish control over execution time to choose whether to perform interval splitting or run the solver for a longer time to improve bounds, or a word-length optimisation framework that performs many repeated calculations of bounds may desire a mix of quick approximate bounds for initial global word-length assignments, and tight bounds for individual word-length optimisations.

The main original contributions of this chapter are as follows:

- a discussion of the trade-off between execution time and quality of bounds for existing analytical methods,
- a description of a technique to control the size of a rational function and hence the execution time to find provable bounds using Handelman representations,
- an explanation of how approximations can be used to bound non-algebraic functions within this framework,
- demonstrations of the use of our approach on simple instances of iterative algorithms to find the solution to a system of linear equations to create hardware with guaranteed error properties, whilst saving 25% slices in comparison to optimising the precision using competing analytical techniques, and almost 80% in comparison to using IEEE double precision.

5.1. Scalability of existing approaches

When attempting to create a tool that can calculate bounds on the range or relative error on a variable in an algorithm, there are two important metrics: quality of bounds and scalability. A tool which can find tight bounds will generally be able to guarantee that hardware will satisfy the chosen output criterion with a lower precision than a tool which calculates wide bounds. Scalability is important because rounding errors arise from every operation and the tool must be able to keep track of all the potential errors that can arise throughout an algorithm and be able to calculate bounds in a tractable time to be of any use. In Chapter 2, we previously have discussed the strengths and weaknesses of the existing methods to calculate such bounds in terms of quality of bounds; in this section, we analyse the worst case execution

time of these approaches in terms of the number of operations in the code, denoted n_o , and the number of input variables n .

Using interval arithmetic, an interval evaluation is performed to find an initial bound on the result of every floating point operation in the code, after which an extra interval evaluation is used to find the bounds taking into account the floating point model of error, where the latter is computed by replacing each of the variables δ_i with an interval. For example, as seen in Chapter 2, computing the bounds for the multiplication operation requires 4 multiplications, along with a min and a max, for every operation in the code; indeed, this is the worst case number of computations for a single operation in interval arithmetic. Consequently, the worst case execution time is proportional to the number of operations, or $O(n_o)$. In Chapter 2, we also highlighted a number of approaches that are used in global interval analysis optimisation tools to improve the results obtained from interval arithmetic, these include automatic differentiation and interval splitting. Because automatic differentiation is based upon interval analysis, this too scales in proportion to the number of operations $O(n_o)$, however, as we mentioned in Chapter 2, interval splitting scales $O(n_o n_s^{n_{sv}})$, where n_s is the number of splits, and n_{sv} is the number of variables that are split. Even though the time will still grow in proportion to the number of operations, the control over this execution time is very limited because every split causes an exponential growth. Interval splitting could be also be used in conjunction with affine arithmetic or Taylor series with interval remainders bounds to trade execution time for tighter bounds, but at a similar cost.

When bounding a variable using affine arithmetic, first an affine polynomial must be created that bounds the range of the every intermediate variable, then the bounds for the polynomial bounding the range of the desired variable are calculated by interval arithmetic. When creating the affine polynomial, the operation that requires the most execution time is often performing any polynomial approximation, however, as we have mentioned in Chapter 2, there are many methods to calculate such bounds which trade quality of approximation for execution time. Furthermore, the number of operations where it is necessary to perform such approximations are likely to be small. If polynomial approximation is performed with minimal computational effort (for example creating a zero order interval approximation), the operation that requires the greatest execution time is once again multiplication. For multiplication, one must first calculate a polynomial resulting from the product of two polynomials, then bound all the

monomials of this polynomial that are of second order or greater. The maximum number of monomials in the product of two polynomials is given by the product of the number of monomials in each polynomial, meaning that the execution time to create the output polynomial is proportional to the number of monomials in the two input polynomials. In affine arithmetic, a variable exists for every input variable and every polynomial approximation. However, using the multiplicative model of floating point error which we have described in Chapter 4, an extra variable bounding the finite precision error will be added after every operation, and because this error is multiplicative, in affine arithmetic, this means it will create at least one second order term which must be bounded with a second new variable. Altogether, this means that the number of monomials in the polynomial bounding a variable in an algorithm is $O(n_o + n)$, the number of monomials in the product of two polynomials is also $O(n_o + n)$, and because this contains second order terms which must be bounded using interval arithmetic, finding a polynomial to bound the result will require $O(n_o + n)$ operations, and the execution time for n_o operations is thus $O(n_o(n_o + n))$.

Taylor series with interval remainder bounds attempt to obtain a level of control over the quality of bounds for scalability trade-off by making it possible for a user to choose the maximum order of the polynomial. However, we note that even in the simplest case, when restricting to first order, the polynomial will suffer from similar problems to affine arithmetic, because even though it uses the interval remainder in the place of an additional variable to bound error, the number of monomials in the polynomial is still of $O(n_o + n)$ and the number of monomials from the product of two polynomials is still $O(n_o(n_o + n))$, meaning this is still not a scalable approach. Furthermore, by allowing higher order terms, for a maximum order ρ , the number of monomials in any polynomial bounding the range of an intermediate variable can grow as $O\binom{n_o + n + \rho}{\rho}$, and hence bounding the higher order terms will require of $O\binom{n_o + n + \rho}{\rho}$ interval evaluations. Furthermore, polynomial approximations using Taylor series become much more complex for higher order polynomials.

This analysis makes it clear that for existing work only interval arithmetic has an execution time that scales well with the number of floating point operations, however, as we have discussed in Chapter 2 and shown in our experiments in Chapter 4, interval arithmetic is unable to find tight bounds due to dependencies between variables. Our aim is to create an approach where the execution time of grows

in proportion with the code size $O(n_o)$ and provides a user a very flexible level of control over the trade-off between execution time per floating point operation and can still obtain bounds approaching the tightness of the approach described in Chapter 4. We discuss the main method to obtain a control over the execution time in Section 5.2, and our overall approach which uses this heuristic to calculate bounds on the range of variables in an algorithm in Section 5.3.

5.2. Controlling the execution time to bound the result of an operation

Affine arithmetic (AA) and Taylor series with interval remainder methods (TwIR) have worst case quadratic execution time with respect to the number of operations in the code because a polynomial bounding the range of every intermediate variable in the code must be created and the number of monomials in each polynomial is proportional to the number of preceding operations in the code. As a result, the basic concept we employ to obtain control over the execution time to bound the result of an operation is to directly control the number of monomials in every polynomial to a user chosen value, N , and hence bound the worst case execution time to create any intermediate polynomial to be some function of N_c . As a result, because the worst case execution time to create any intermediate polynomial becomes constant, the overall execution time of our algorithm grows as $O(n_o)$, and the choice of N_c provides the user the ability to trade potential tightness of bounds with execution time. We note that this level of control is a much finer level of control than Taylor forms. The algorithm we employ to perform this is given in Figure 5.1, in the rest of this section, we describe the rationale behind this algorithm.

$(\hat{p}, k) = \text{Simplify Polynomial } (p, N, k)$
 1: $\hat{p} = N$ monomials from p with the largest magnitude of potential contribution to final bounds, calculated by IA
 2: $C_k + \zeta_k$ = new polynomial bounding potential contribution of other monomials in p
 3: $\hat{p} = \hat{p} + C_k + \zeta_k$
 4: $k = k + 1$

Figure 5.1.: Algorithm to control the size of the polynomial.

The main concept behind this algorithm is that many monomials within a polynomial represent a small contribution towards the final bounds of the function, and hence if the dependency information of these monomials is lost, it has little impact on the final result. As such, this algorithm retains the monomials that have the greatest potential contribution to the final bounds, calculated by computing the

Table 5.1.: Potential contribution of each monomial in $(1 + x_1)(1 + y_1)(1 + \delta_1)$.

Compute $a = x \bullet y$, where $x = [0.8; 1.2]$, $y = [0.9; 1.1]$ in 6 bit floating point let $ x_1 \leq 0.2$, $ y_1 \leq 0.1$, $ \delta_i \leq 2^{-6} \Rightarrow x \in 10(1 + x_1)$, $y \in 10(1 + y_1)$ $a = (1 + x_1)10(1 + y_1)(1 + \delta_1)$ $= (100 + 100x_1 + 100y_1 + 100\delta_1 + 100x_1y_1 + 100x_1\delta_1 + 100y_1\delta_1 + 100x_1y_1\delta_1)$			
Monomial	Potential Contribution	Monomial	Potential Contribution
100	100	$100x_1$	± 20
$100\delta_1$	± 0.09765625	$100x_1\delta_1$	± 0.01953125
$100y_1$	± 10	$100x_1y_1$	± -2
$100y_1\delta_1$	± 0.009765625	$100x_1y_1\delta_1$	± 0.001953125

bounds of each monomial using interval arithmetic, and then bounds the remaining monomials using a polynomial consisting of a constant C_i a single monomial ζ_i .

While TwIR has a similar motivation, it retains the lower order monomials and bounds the higher order monomials, with a user choosing the maximum order. In contrast, our algorithm retains the monomials which have the greatest potential contribution to the final bounds and bounds the remaining monomials, with a user choosing the number of monomials. This approach is capable of returning better bounds because while higher order monomials will in general have a lower potential contribution to the final bounds of the polynomial, this is not always the case. We demonstrate this using Table 5.1, which states the potential contributions to the final bounds, computed using interval arithmetic, for all the individual monomials when computing bounds for a simple problem. In this table, the second order monomial x_1y_1 has a higher contribution than the first order monomial δ_1 . Because some input variables may have much wider ranges than other input variables, and often wider ranges than variables bounding finite precision errors, this approach is logical as it is most important to retain the dependency information for the variables with the widest bounds whereas the dependency information for small perturbations can be sacrificed in favor of a reduced execution time. In the final stage, we represent the range of all unwanted monomials using a monomial ζ_i along with a constant C_i that centers the monomial ζ_i over the desired range, for this has previously been shown to obtain the best error properties [130]. We note that alternate level of control would be to select all monomials that contribute up to a certain percentage of the largest contribution; this would provide a better control over final quality of bounds at the cost of a weaker control over the execution time, because it would be unclear how many monomials would be kept at any given time.

Once we have bounded all unwanted monomials, we represent them using a centered variable ζ_i ,

similar to AA, and unlike TwIR. This is to avoid the problem that arises in TwIR when multiplying an interval remainder bound I_ρ by a polynomial T_ρ . Whenever this occurs, we must first bound the polynomial T_ρ before calculating the product of these two intervals and bounding this polynomial can result in a loss of dependency information. For example, in the case of multiplication, given in equation (5.1), all correlation between $T_1(x)I_2$ and $T_1(x)T_2(x)$ is lost. This can contribute to bounds that are significantly worse than IA.

$$(T_1(x) + I_1)(T_2(x) + I_2) = (T_1(x)T_2(x) + T_1(x)I_2 + T_2(x)I_1 + I_1I_2). \quad (5.1)$$

5.3. Creating a scalable model bounding the range of variable in finite precision arithmetic

While the above algorithm given in Figure 5.1 would be sufficient to control the execution time if integrated into either affine arithmetic or Taylor series with interval remainder bounds, a combined approach would still suffer from dependencies within a polynomial. Furthermore, as we have shown in Chapter 4, using polynomial representations means that divisions must be approximated with a polynomial, and this can lead to wider bounds. In this section, we describe how we combine this algorithm with the technique to bound rational functions using Handelman representations which we described in Chapter 4 to take advantage of the ability to handle dependencies within a polynomial or rational function and create a scalable framework to find tight bounds for the range or relative error for any variable in an algorithm.

5.3.1. Representing the range of a variable

One of the problems with using the polynomial simplification algorithm described in Figure 5.1 is that when we replace all of the monomials with small contributions to the final bounds with a new monomial, we lose information on whether those monomials with small contributions were a function of only the input variables, or a function of both input variables and finite precision errors. This is an issue when computing bounds on the relative error. To compute the relative error, if we have a polynomial p which bounds the range in infinite precision, and a polynomial $\hat{p}$ which bounds the range in the presence of

finite precision errors, then the bound on the relative error is computed by maximizing the function $|\frac{p-\hat{p}}{p}|$. However, if we control the size of the polynomials p and $\hat{p}$ using the algorithm described in Figure 5.1, we would lose any correlation between the added monomials bounding small contributions in p and $\hat{p}$.

To demonstrate how this can become a problem, we use a simple example shown in Table 5.2. In this example, we attempt find bounds on the relative error of the computation $(x \bullet y) \bullet z$, where $x \in [0.8; 1.2]$, $y \in [0.9; 1.1]$, $z \in [9.9; 10.1]$ using a 6-bit precision, where we limit the maximum number of monomials in a polynomial to be 6. If we were to compute the relative error of this operation, according to Table 5.2, we must compute bounds on the function $|\frac{z_1 + 2.048\zeta_2 - 10\delta_1 - 25.3203\zeta_3}{10 + 10x_1 + 10y_1 + z_1 + 10x_1y_1 + 2.048\zeta_2}|$. The problem with this is that there is correlation between the monomials z_1, ζ_2 and ζ_3 which is lost due to the simplification algorithm. This leads to much wider bounds on relative error.

Table 5.2.: Controlling polynomial size using the algorithm defined in Figure 5.1 to control polynomial size with $N = 6$.

Calculate the relative error of the computation $(x \bullet y) \bullet z$, where $x \in [0.8; 1.2]$, $y \in [0.9; 1.1]$, $z \in [9.9; 10.1]$ in floating point with a 6-bit mantissa. Let $ x_1 \leq 0.2, y_1 \leq 0.1, z_1 \leq 0.1 \Rightarrow x = (1 + x_1), y = (1 + y_1), z = (10 + z_1)$ Also let $\forall_i, \delta_i \leq 2^{-6}, \zeta_i \leq 2^{-6}$			
Create polynomials to bound the range of every intermediate variable			
Code	Polynomial bounding variable range	Polynomial in canonical form	Simplified polynomial
$a = x \bullet y$	$a = \frac{(1 + x_1)(1 + y_1)}{(1 + x_1)(1 + y_1)(1 + \delta_1)}$	$\frac{1 + x_1 + y_1 + x_1y_1}{1 + x_1 + y_1 + x_1y_1 + \delta_1 + x_1\delta_1 + y_1\delta_1 + x_1y_1\delta_1}$	$\frac{1 + x_1 + y_1 + x_1y_1}{1 + x_1 + y_1 + x_1y_1 + \delta_1 + 0.32\zeta_1}$
$b = a \bullet z$	$b = \frac{(1 + x_1 + y_1 + x_1y_1)(10 + z_1)}{(1 + x_1 + y_1 + x_1y_1 + \delta_1 + 0.32\zeta_1)(10 + z_1)(1 + \delta_2)}$	$\frac{10 + 10x_1 + 10y_1 + 10x_1y_1 + z_1 + x_1z_1 + y_1z_1 + x_1y_1z_1}{10 + 10x_1 + 10y_1 + 10x_1y_1 + 10\delta_1 + 3.2\zeta_1 + z_1 + x_1z_1 + y_1z_1 + x_1y_1z_1 + z_1\delta_1 + 0.32z_1\zeta_1 + 10\delta_2 + 10x_1\delta_2 + 10y_1\delta_2 + 10x_1y_1\delta_2 + 10\delta_1\delta_2 + 3.2\zeta_1\delta_2 + z_1\delta_2 + x_1z_1\delta_2 + y_1z_1\delta_2 + x_1y_1z_1\delta_2 + z_1\delta_1\delta_2 + 0.32z_1\zeta_1\delta_2}$	$\frac{10 + 10x_1 + 10y_1 + z_1 + 10x_1y_1 + 2.048\zeta_2}{10 + 10x_1 + 10y_1 + 10\delta_1 + 10x_1y_1 + 25.3203\zeta_3}$
Find relative error of every intermediate variable			
Variable	Rational function bounding relative error		Bound on relative error using IA
$ \frac{a - \hat{a}}{a} $	$ \frac{\delta_1 + 0.32\zeta_1}{1 + x_1 + y_1 + x_1y_1} $		0.0303
$ \frac{b - \hat{b}}{b} $	$ \frac{z_1 + 2.048\zeta_2 - 10\delta_1 - 25.3203\zeta_3}{10 + 10x_1 + 10y_1 + z_1 + 10x_1y_1 + 2.048\zeta_2} $		0.1026

In order to avoid this problem, we separate a polynomial $\hat{p}$ into the sum of two polynomials $p + p_e$, where the polynomial p consists of monomials that are only a function of the input variables, and the polynomial p_e store the additional monomials resulting from the introduction of finite precision errors. We note now that even if we apply the polynomial simplification algorithm, the polynomial p will bound

the result in infinite precision, and by keeping these polynomials separate, we can now compute bounds on the relative error by finding bounds to the rational function $|\frac{p_\epsilon}{p}|$; this allows us to find much tighter bounds, as shown in Table 5.3.

Table 5.3.: Controlling polynomial size using the algorithm defined in Figure 5.1 with $N = 3$ to control separate polynomials for range in infinite precision and the additional monomials resulting from finite precision errors.

Calculate the relative error of the computation $(x \bullet y) \bullet z$, where $x \in [0.8; 1.2], y \in [0.9; 1.1], z \in [9.9; 10.1]$ in floating point with a 6-bit mantissa. Let $ x_1 \leq 0.2, y_1 \leq 0.1, z_1 \leq 0.1 \Rightarrow x = (1 + x_1), y = (1 + y_1), z = (10 + z_1)$. Also let $\forall_i, \delta_i \leq 2^{-6}, \zeta_i \leq 2^{-6}$			
Create polynomials to bound the range of every intermediate variable			
Code	Polynomial bounding range of variable	Simplified polynomial, p , bounding range in infinite precision	Simplified polynomial, p_ϵ , bounding finite precision errors
$a = x \bullet y$	$(1 + x_1)(1 + y_1)(1 + \delta_1)$	$1 + x_1 + 7.68\zeta_1$	$\delta_1 + x_1\delta_1 + 0.12\zeta_2$
$b = a \bullet z$	$(1 + x_1 + 7.68\zeta_1 + \delta_1 + x_1\delta_1 + 0.12\zeta_2)(10 + z_1)(1 + \delta_2)$	$10 + 10x_1 + 83.9680\zeta_3$	$1.2\zeta_2 + \delta_2 + 15.6723\zeta_4$
Find relative error of every intermediate variable			
Variable	Rational function bounding relative error		Bound on relative error using IA
$ \frac{a_\epsilon}{a} $	$ \frac{\delta_1 + x_1\delta_1 + 1.12\zeta_2}{1 + x_1 + 7.68\zeta_1} $		0.0244
$ \frac{b_\epsilon}{b} $	$ \frac{1.2\zeta_2 + \delta_2 + 15.6723\zeta_4}{10 + 10x_1 + 83.9680\zeta_3} $		0.0363

While this technique is an effective method to describe polynomials, in order to take advantage of the ability to retain correlation between the numerator and denominator of Handelman representations, avoiding the problem of polynomial approximation of division which we described in Chapter 4, we bound the range of any intermediate variable in the code using a rational function of the form $\frac{n+n_\epsilon}{d+d_\epsilon}$, where n and d are the numerator and denominator polynomials which contribute to the bounds in infinite precision, and n_ϵ and d_ϵ store the additional monomials which result from the introduction of finite precision errors.

5.3.2. Bounding the range of variables in finite precision arithmetic for a user algorithm

In order to compute bounds on the range or relative error for any variable within an algorithm, we first compile the target algorithm into a 2-input static single assignment (SSA) intermediate representation consisting of vector operations, with the aid of techniques such as loop unrolling. Throughout our algorithms, we prefer to operate on vectors in order to take advantage of the fact that every element in a vector will typically share the same denominator and hence our algorithms are designed to retain this correlation so as to improve the tightness of bounds. We then proceed to calculate bounds on this

```

Bound variable in code ( $N_1, N_2, \text{code}$ ).
//  $N_1, N_2$  are user chosen variables to control the maximum polynomial sizes
// We denote vectors of rational functions bounding variables with  $\mathbf{v}$  where the  $i^{th}$  rational function of this vector is indexed  $v^i$ .
// The total number of elements in a vector is given by  $|\mathbf{v}|$ . As a scalar variable is a vector consisting of only one element so we omit the superscript  $i$ .
// Number of variables  $\delta_j, \zeta_k$  are determined at run time.
1: Create set  $V$  of all input variables as vectors of the form:
   
$$\mathbf{v}_a = \left[ \frac{n_a^1 + n_{a\epsilon}^1}{d_a^1 + d_{a\epsilon}^1}, \dots, \frac{n_a^{|\mathbf{v}_a|} + n_{a\epsilon}^{|\mathbf{v}_a|}}{d_a^{|\mathbf{v}_a|} + d_{a\epsilon}^{|\mathbf{v}_a|}} \right].$$

2:  $(j, k) = (1, 1)$ .
3: for every operation  $\mathbf{v}_a \odot \mathbf{v}_b$  in intermediate representation do
4:    $(\mathbf{v}_\star, j) = \text{Compute rational function}(\mathbf{v}_a, \odot, \mathbf{v}_b, N_1, N_2, j)$ 
5:   for  $i = 1$  to  $|\mathbf{v}_\star|$  do
6:      $(n_\star^i, n_{\star\epsilon}^i, k) = \text{Simplify Polynomials}(n_\star^i, n_{\star\epsilon}^i \delta_j + n_{\star\epsilon}^i (1 + \delta_j), N_1, N_2, k)$ 
7:      $j = j + 1$ 
8:   end for
9:    $(d_\star, d_{\star\epsilon}, k) = \text{Simplify Polynomials}(d_\star, d_{\star\epsilon}, N_1, N_2, k)$ 
10:   $j = j + 1$ 
11:   $(\mathbf{v}_\star) = \text{Normalise Coefficients}(\mathbf{v}_\star)$ .
12:  Add  $\mathbf{v}_\star$  to  $V$ 
13: end for
14: Bound desired variable  $\mathbf{v}$  in  $V$  using Handelman representations
 $(\hat{p}, \hat{p}_\epsilon, k) = \text{Simplify Polynomials}(p, p_\epsilon, N_1, N_2, k)$ 
1:  $(\hat{p}, k) = \text{Simplify Polynomial}(p, N_1, k)$ 
2:  $(\hat{p}_\epsilon, k) = \text{Simplify Polynomial}(p_\epsilon, N_2, k)$ 
    
```

Figure 5.2.: Overall algorithm to find bounds on the range or relative error of a variable from a user input code.

intermediate representation using a set of simple algorithms summarised in Figures 5.2, 5.3 and 5.4. In the rest of this section, we explain the rationale behind each of these algorithms.

Bound variable in code

In the main algorithm, we first create a set V containing all the input variables in the algorithm, stored as vectors wherever this is applicable. Our algorithm proceeds by sequentially examining each vector operation in the intermediate representation and creates new rational functions which bound the range of every output element from this operation. The operations we support are scalar multiplication, scalar division, scalar addition and subtraction, vector addition and subtraction, dot products, and any other function to which a polynomial approximation can be computed. The reason we prefer to perform vector operations is because after creating each new rational function, the number of monomials in the polynomials n and d are controlled according to a user choice N_1 , and the number of monomials in the polynomials n_ϵ and d_ϵ are controlled according to a user choice N_2 , where the choice of N_1 and N_2 is left to a user to trade the execution time against the potential quality of bounds, using the

algorithm in Figure 5.1. If we were to perform each operation on scalars instead of performing a single vector operation, then if the number of monomials in the polynomials d and d_ϵ created by the operation are greater than N_1 or N_2 , then the denominator polynomials would be simplified with the use of a different variable ζ_k for each element in the vector, meaning that none of the denominators for the vector would be the same. By performing a single vector operation, we only need to simplify the denominator polynomial for the entire vector once, retaining correlation for all the vector elements and improving the overall bounds. We note that because each numerator polynomial for a vector will in general be different, each of these are simplified individually, and in order to capture round-off uncertainty, we first apply the desired model error to the polynomial n_ϵ , as defined in Chapter 4. In Figures 5.2 and 5.3, we have used the multiplicative model for floating point error which we have described in Chapter 4. After performing any simplification, the coefficients between the numerator and denominator are normalised to prevent either becoming unnecessarily large. Finally, once we have created a rational function to bound the range of the desired output variable, we calculate bounds using Handelman representations, as described in Chapter 4, to try to find the best bounds taking into account any dependencies in the rational function.

Compute rational function

Figure 5.3 describes how we create a rational functions which bound the range of every intermediate variable in the SSA version of the code. In general, a rational function representing the result of the basic algebraic operations ($\odot \in \{+, -, *, /\}$) applied to two input rational functions can easily be computed symbolically. For example, equation (5.2) shows how to perform multiplication of two rational functions $v_a = \frac{n_a + n_{a\epsilon}}{d_a + d_{a\epsilon}}$ and $v_b = \frac{n_b + n_{b\epsilon}}{d_b + d_{b\epsilon}}$; in this equation, we have used brackets to separate the polynomials which consist of both input variables and finite precision errors. However, for addition or subtraction when the denominator polynomials are different for the two operands, we apply a different approach. The reason for this exception is that this operation can result in a very large numerator polynomial which has lots of correlation with the denominator polynomial, as shown in equation (5.3). However, when we subsequently simplify the numerator and denominator polynomials, according to the algorithm in Figure 5.2, we lose correlation between these polynomials and because the number of monomials in the

```

(v*, j) = Compute rational function (va, ⊙, vb, N1, N2, j)
1: if (⊙ == •) then
2:   (d*, d*ε) = (dadb, dadbε + daεdb + daεdbε)
3:   if (|vb| == 1) then
4:     for i = 1 to |va| do
5:       (n*i, n*εi) = (n̂ + nainb, n̂ε + nainbε + naεinb + naεinbε)
6:     end for
7:   else
8:     (n*, n*ε) = (0, 0)
9:     for i = 1 to |va| do
10:      (n*, n*ε) = (n* + nainbi, n*ε + nainbεi + naεinbi + naεinbεi
                    + δj(nainbi + nainbεi + naεinbi + naεinbεi))
11:      n*ε = n̂δj + n̂ε(1 + δj+1)
12:      j = j + 2
13:    end for
14:  end if
15: else if ((⊙ == +) or (⊙ == -)) then
16:   if (da, daε) == (db, dbε) then
17:     for i = 1 to |va| do
18:       if (|vb| == 1) then
19:         (n̂i, n̂εi) = (nai ⊙ nb, naεi ⊙ nbε)
20:       else
21:         (n̂i, n̂εi) = (nai ⊙ nbi, naεi ⊙ nbεi)
22:       end if
23:     end for
24:   (d*, d*ε) = (da, daε)
25:   else
26:     v1 =  $\frac{d_a + d_{aε}}{1}$ , v2 =  $\frac{d_b + d_{bε}}{1}$ 
27:     v1 = Compute Polynomial Approximation(v1, λx.x-1)
28:     v2 = Compute Polynomial Approximation(v2, λx.x-1)
29:     (v1, j) = Compute rational function (va, •, v1, N1, N2, j)
30:     (v2, j) = Compute rational function (vb, •, v2, N1, N2, j)
31:     (v*, j) = Compute rational function (v1, ⊙, v2, N1, N2, j)
32:   end if
33: else if (⊙ == ÷) then
34:   vb =  $\frac{d_b + d_{bε}}{n_b + n_{bε}}$ 
35:   (v*, j) = Compute rational function (va, •, vb, N1, N2, j)
36: else
37:   v* = Compute Polynomial Approximation(va, ⊙)
38: end if
    
```

Figure 5.3.: Algorithm to create rational functions bounding intermediate variables.

numerator is substantially larger than in the denominator.

$$\frac{n_a + n_{aε}}{d_a + d_{aε}} \times \frac{n_b + n_{bε}}{d_b + d_{bε}} = \frac{n_a n_b + (n_a n_{bε} + n_b n_{aε} + n_{aε} n_{bε})}{d_a d_b + (d_a d_{bε} + d_b d_{aε} + d_{aε} d_{bε})} \quad (5.2)$$

$$\frac{n_a + n_{aε}}{d_a + d_{aε}} + \frac{n_b + n_{bε}}{d_b + d_{bε}} = \frac{n_a d_b + n_b d_a + (n_a d_{bε} + n_b d_{aε} + n_{aε} d_{bε} + n_{bε} d_{aε} + n_{aε} d_{bε} + n_{bε} d_{aε})}{d_a d_b + (d_a d_{bε} + d_b d_{aε} + d_{aε} d_{bε})} \quad (5.3)$$

As a result, in Figure 5.3, we instead compute a rational function to bound the result by first normalis-

ing the denominators of the two input rational functions to 1 by multiplying their numerator polynomials by a polynomial approximation of the reciprocal of their denominator polynomials. Though we note that this can lead to our approach finding wider bounds than the true bounds due to the errors in the approximation, experimentally these errors have in general been found to be smaller than applying our algorithm without this additional approximation.

To implement other elementary functions, such as ($\odot \in \{\text{sqrt}(), \sin(), \exp()\}$), as well as approximating the reciprocal, we could use any of the well known techniques in approximation theory such as Taylor approximations, Chebyshev approximations, or the Remez algorithm, as we have described in Chapter 2 to create polynomial or rational function approximations. In general, the choice of approximation will trade quality of bounds with execution time, and due to the wealth of research in this area, this is left to a user to choose the optimum approximation. However, we note that this flexibility is unavailable when using AA and TwIR, for these require the polynomial approximation to be of a specific form. Furthermore, because if we approximate a function over a smaller range, the approximation will generally have less worst case error, we use Handelman representations to find bounds on the range of a variable before performing any polynomial approximation.

Normalise Coefficients

$v_{\star}$ = Normalise Coefficients ($\hat{\theta}$).

- 1: Find $c_1 = \lfloor \log_2(\text{largest coefficient}(\hat{n}, \hat{n}_{\epsilon})) \rfloor$
- 2: Find $c_2 = \lfloor \log_2(\text{largest coefficient}(\hat{d}, \hat{d}_{\epsilon})) \rfloor$
- 3: Find $c_{gcd} = \text{greatest common divisor}(c_1, c_2)$
- 4: $n_{\star\epsilon} = \hat{n} * 2^{-c_{gcd}}$
- 5: $n_{\star\epsilon} = \hat{n}_{\epsilon} * 2^{-c_{gcd}}$
- 6: $d_{\star} = \hat{d} * 2^{-c_{gcd}}$
- 7: $d_{\star\epsilon} = \hat{d}_{\epsilon} * 2^{-c_{gcd}}$

Figure 5.4.: Algorithm to normalise coefficients.

The final step in our algorithm is to normalise the coefficients in the numerator and denominator polynomials. The reason this stage is necessary is that by keeping the numerator and denominator polynomials separate in a rational function, any cancellation between these polynomials does not occur, meaning that over time the coefficients of these two monomials can get excessively large. We therefore divide both these polynomials by an appropriate factor to avoid this problem; this factor is a power of

two to ensure the division is performed without error.

5.4. Results

5.4.1. Benchmarks

We have created two benchmarks, shown in Figures 5.5 and 5.6, to help demonstrate the benefits of our proposed approach in terms of scalability and quality of bounds in comparison to all the main competing methods which are capable of bounding error: IA, TwIR, AA, and Handelman representations. In these figures, we present the original pseudo code alongside a breakdown of this pseudo code into vector operations, because this in the original code, no order is specified, but the order of operations affects the accumulation of errors, and hence this information is required to calculate any bound on the range or relative error. In the first test, because the widest input intervals are for the diagonal elements of the matrix A , and the input vector b in the first test, we also examine the impact of using interval splitting on these elements when finding bounds using IA as splitting these intervals will have the greatest impact on the final bounds. We do not perform this for the second test because it is unclear which variables would be best to split, whilst splitting all variables, as we shall see in Section 5.4.2, will require too large an execution time. These benchmarks are large compared to similar publications in the field; in contrast, the examples of [84] consist of up to 10 input variables and does not taking finite precision errors into account.

In the first test, because there is little error in the first order approximation of $1/a_{i,i}$, and all the non-affine operations are multiplications of polynomials or rational functions x_i by input variables $a_{i,j}$, the majority of the information regarding the final range of the values x_i is in the first order terms. This implies IA, AA and TwIR should be able to calculate tight bounds, so in this test, we wish to demonstrate that our approach will perform well where the existing methods ought to perform well. In contrast, the second test involves products of polynomials or rational functions bounding intermediate variables, the division of a multivariate polynomial and a square root operation, and hence we wish to demonstrate that our approach can perform well in a more complex problem.

When performing polynomial approximations, because the focus of this paper is on the method to

$$A = \begin{pmatrix} 100 & -10 & -15 & -4 & 16 \\ -10 & 105 & -13 & 4 & 14 \\ -14 & -13 & 90 & 19 & -11 \\ 12 & 4 & 14 & 110 & 15 \\ 16 & 14 & -10 & 8 & 95 \end{pmatrix} \pm 1\% b = \begin{pmatrix} 200 \\ -120 \\ -160 \\ 180 \\ -100 \end{pmatrix} \pm 1,$$

```

// The  $i^{th}$  element of a vector  $v$  is indexed  $v^i$  as before
// The  $j^{th}$  row of a matrix  $A$  is indexed  $A(j)$ 
1: for  $k = 1; k \leq 8; k++$  do
2:   for  $j = 1; j \leq 5; j++$  do
3:      $x^j = (1 - w)x^j + \frac{w}{A(j)j} (b_j - \sum_{i=1, i \neq j}^5 A(j)^i x^i)$ 
4:   end for
5: end for
    
```

(a) 5x5 Successive over relaxation iteration pseudo code [5].

```

// The  $i^{th}$  element of a vector  $v$  is indexed  $v(i)$ 
// The element in column  $i$ , row  $j$  of a matrix  $A$  is indexed  $A(i, j)$ 

1: // Initialisations
2: for  $i = 1; i \leq 5; i++$  do
3:    $wDIVa.i = w/A(i)^i$ 
4:   for  $j = 1; j \leq 5; j++$  do
5:      $ASUBdiagA.i^j = A(j)^i$ 
6:   end for
7:    $ASUBdiagA.i^i = 0$ 
8: end for
9:  $USUBw = 1 - w$ 
10: // Algorithm
11: for  $k = 1; k \leq 8; k++$  do
12:   for  $j = 1; j \leq 5; j++$  do
13:      $ax = ASUBdiagA^j \bullet x$ 
14:      $bSUBax = b^j - ax$ 
15:      $rhs = wDIVa^j \bullet bSUBax$ 
16:      $lhs = USUBw \bullet x^j$ 
17:      $x^j = lhs - rhs$ 
18:   end for
19: end for
    
```

(b) 5x5 Successive over relaxation iteration expressed using vector operations.

Figure 5.5.: 5x5 Successive over relaxation benchmark.

find bounds for a variable within an algorithm, as opposed to approximation theory, in order to minimise the effect of the quality of this approximation, we use the same polynomial approximation method for affine arithmetic and our approach, with the exception of using Handelman representations to find the input range for this approximation, as mentioned in Section 5.3.2.

5.4.2. Test 1: Successive over relaxation

Scalability: Figures 5.7(a) and 5.7(b) demonstrate how the execution time on an Intel Xeon E5345 of each of the methods grows with the number of operations when computing the range or relative error for intermediate variables over the course of the successive over relation algorithm. Table 5.4, states the final time for each of the approaches to compute the bounds on range and relative error.

For the range analysis case, seen in Figure 5.7(a), IA, 1st order TwIR, AA and our approach initially scale well with the number of operations, whereas TwIR of orders greater than 1 and Handelman representations scale poorly due to exponential growth, as mentioned in Chapter 4. However, it is also clear that only IA and our approach have an execution time proportional to the number of operations, as

$$A = \begin{pmatrix} 50 & -60 & 70 & -80 \\ -60 & -60 & 40 & 70 \\ 70 & 40 & 40 & -30 \\ -80 & 70 & -30 & -40 \end{pmatrix} \pm 0.25, \quad x_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad b = \begin{pmatrix} 80 \\ 60 \\ 40 \\ 70 \end{pmatrix} \pm 0.25$$

```

1:  $v_1 = b - Ax_0$ ;
2:  $\beta_1, \eta = \|v_1\|_2$ 
3:  $v_0, w_0, w_{-1} = [0000]'$ 
4:  $\sigma_0, \sigma_1 = 0; \gamma_0, \gamma_1 = 1$ 
5: for  $i = 1; i \leq 3; i++$  do
6:    $v_i = \frac{v_i}{\beta_i}$ 
7:    $\alpha = v_i^T Av_i$ 
8:    $v_{i+1} = Av_i - \alpha v_i - \beta v_{i-1}$ 
9:    $\beta_{i+1} = \|v_{i+1}\|_2$ 
10:   $\delta = \gamma_i \alpha_i - \gamma_{i-1} \sigma_i \beta_i$ 
11:   $\rho_1 = \sqrt{\delta^2 + \beta_{i+1}^2}$ 
12:   $\rho_2 = \sigma_i \alpha_i + \gamma_{i-1} \gamma_i \beta_i$ 
13:   $\rho_3 = \sigma_{i-1} \beta_i$ 
14:   $\gamma_{i+1} = \frac{\delta}{\rho_1}$ 
15:   $\sigma_{i+1} = \frac{\beta}{\rho_1}$ 
16:   $w_i = \frac{v_i - \rho_3 w_{i-2} - \rho_2 w_{i-1}}{\rho_1}$ 
17:   $x_i = x_{i-1} + \gamma_{i+1} \eta w_i$ 
18:   $\eta = -\sigma_{i+1} \eta$ 
19: end for
    
```

(a) MINRES algorithm pseudo code [55].

```

1: // Initialisations
2: for  $i = 1; i \leq 4; i++$  do
3:    $Ax^i = A(i) \bullet x_0$ 
4:    $v_0^i = 0$ 
5:    $w_0^i = 0$ 
6:    $w_{-1}^i = 0$ 
7: end for
8:  $v_1 = b - Ax$ ;
9:  $tmp\_beta = v_1^T \bullet v_1$ 
10:  $\beta_1 = \sqrt{tmp\_beta}$ 
11:  $\sigma_0, \sigma_1 = 0; \gamma_0, \gamma_1 = 1$ 
12: // Algorithm
13: for  $i = 1; i \leq 3; i++$  do
14:    $v_i = \frac{v_i}{\beta_i}$ 
15:   for  $j = 1; j \leq 4; j++$  do
16:      $Av^j = A(j) \bullet v_i$ 
17:   end for
18:    $\alpha = v_i \bullet Av$ 
19:    $\alpha V = \alpha \bullet v_i$ 
20:    $\beta V_{-1} = \beta_i \bullet v_{i-1}$ 
21:    $tmp\_v_{i+1} = \alpha V - \beta V_{-1}$ 
22:    $v_{i+1} = Av - tmp\_v_{i+1}$ 
23:    $tmp\_beta = v_1 \bullet v_1$ 
24:    $\beta_{i+1} = \sqrt{tmp\_beta}$ 
25:    $tmp1\_delta = \gamma_i \bullet \alpha_i$ 
26:    $tmp2\_delta = \gamma_{i-1} \bullet \sigma_i$ 
27:    $tmp2\_delta = tmp2\_delta \bullet \beta_i$ 
28:    $\delta = tmp1\_delta - tmp2\_delta$ 
29:    $tmp\_rho_1 = \delta \bullet \delta$ 
30:    $tmp\_rho_1 = tmp1\_rho_1 + tmp\_beta$ 
31:    $\rho_1 = \sqrt{tmp\_rho_1}$ 
32:    $tmp1\_rho_2 = \sigma_i \bullet \alpha_i$ 
33:    $tmp2\_rho_2 = \gamma_{i-1} \bullet \gamma_i$ 
34:    $tmp2\_rho_2 = tmp2\_rho_2 \bullet \beta_i$ 
35:    $\rho_2 = tmp1\_rho_2 + tmp2\_rho_2$ 
36:    $\rho_3 = \sigma_{i-1} \bullet \beta_i$ 
37:    $\gamma_{i+1} = \frac{\delta}{\rho_1}$ 
38:    $\sigma_{i+1} = \frac{\beta_i}{\rho_1}$ 
39:    $tmp1\_w = \rho_3 \bullet w_{i-2}$ 
40:    $tmp2\_w = \rho_2 \bullet w_{i-1}$ 
41:    $tmp2\_w = tmp1\_w - tmp2\_w$ 
42:    $tmp1\_w = v_i - tmp2\_w$ 
43:    $w_i = \frac{tmp1\_w}{\rho_1}$ 
44:    $tmp\_x = \eta \bullet w_i$ 
45:    $tmp\_x = \gamma_{i+1} \bullet tmp\_x$ 
46:    $x_i = x_{i-1} + tmp\_x$ 
47:    $\eta = -\sigma_{i+1} \bullet \eta$ 
48: end for
    
```

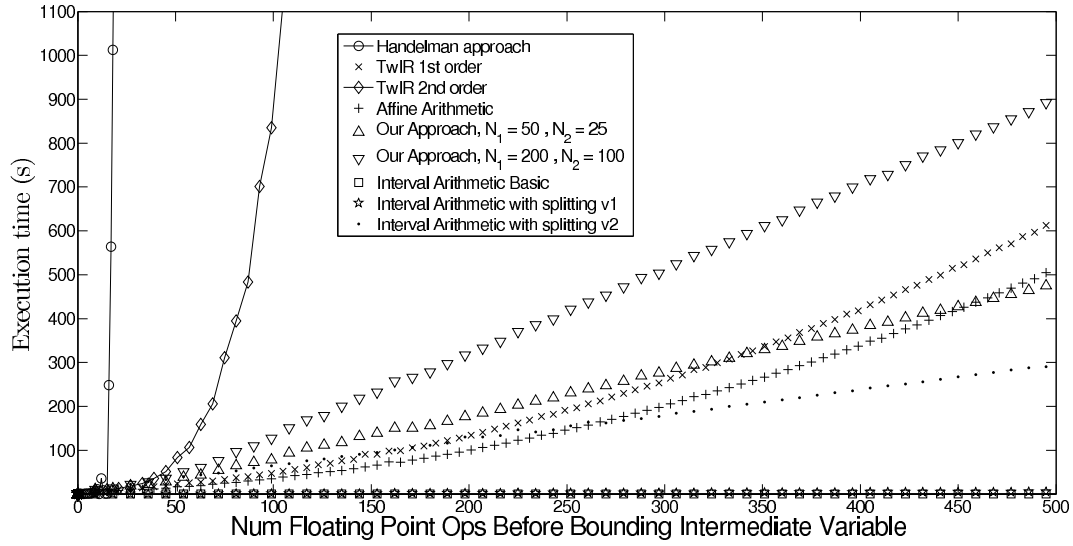
(b) MINRES Algorithm in terms of two input vector operations.

Figure 5.6.: MINRES algorithm benchmark.

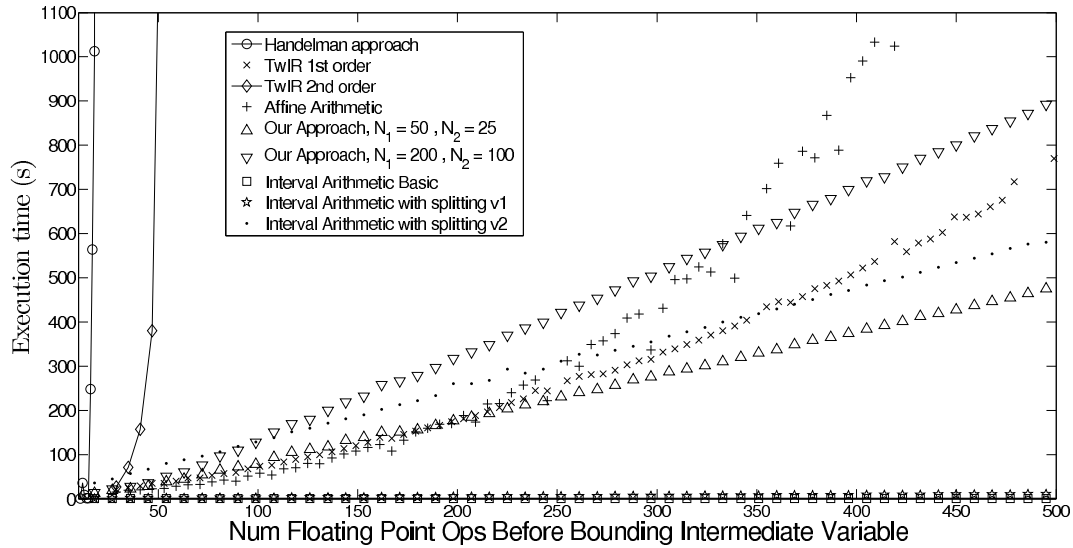
expected from our analysis in Section 5.1, and this means that as the number of operations gets large, our approach can run faster than AA and TwIR. As we have previously mentioned, this is because our approach directly controls the size of the polynomial, whereas the polynomials bounding the range of intermediate variables using TwIR and AA are proportional to the number of operations because a variable bounding the roundoff error is added after every operation. Furthermore, as expected from our analysis in Section 5.1, AA scales worse than 1st order TwIR because it gains an extra variable for every operation as a result of bounding the error of the higher order terms created by the multiplicative model of error. Finally, this graph shows that IA splitting can be used to trade-off execution time for quality of bounds, with the execution time still proportional to the number of operations. However, we note that the level of control over this execution time is very limited. We have calculated bounds using IA without splitting, IA where the chosen variables are split into two regions (IA with splitting v1), and IA where the chosen variables are split into three regions (IA with splitting v2); there is a significant difference in execution time between these approaches, indeed this difference can be several orders of magnitude, as seen in Table 5.4. This is because the IA with splitting v1 requires 2^{10} separate interval evaluations, IA with splitting v2 requires 3^{10} interval evaluations. Clearly, performing any further splits is not scalable, and we note that this is with an intelligent selection of 10 variables to split; a naïve approach of splitting every variable would scale far worse than this approach and is unlikely to find much tighter bounds.

The number of monomials in the AA representation for a given variable is an even larger problem when bounding the relative error of variables after many operations, as seen in Figure 5.7(b). The cause of this is that to compute the relative error, one must first generate two polynomials, a polynomial p representing the range of the desired variable in the absence of error, and a polynomial $\hat{p}$ representing the range in the presence of error, then compute bounds of the function $|\frac{p-\hat{p}}{p}|$. To bound this using AA or TwIR, one must first compute a polynomial approximation of $\tilde{p} = 1/p$ then bound the result of the computation $(p - \hat{p}) \times \tilde{p}$, and because $p, \hat{p}$ and $\tilde{p}$ are large polynomials where the number of monomials in these polynomials are proportional to the number of operations, the result will be a very large polynomial with many monomials that must be bounded.

Quality of bounds: Figures 5.8(a) and 5.8(b) show the bounds on the average range and relative error over each of the output variables (the x-vector) for the example in Figure 5.5 for each of the methods



(a) Execution time of various methods to bound the range of the intermediate variable after a given number of operations within a 5x5 Successive Over Relaxation.



(b) Execution time of various methods to bound the relative error of the intermediate variable after a given number of operations within a 5x5 Successive Over Relaxation.

Figure 5.7.: Range and relative error of various methods to bound error applied to a 5x5 successive over relaxation.

Table 5.4.: Comparison of execution times to compute average relative error of x variables for a given precision.

Method	Average time to compute range of x (s)	Average time to compute relative error of x (s)
Interval Arithmetic Basic	0.01	0.026
Interval Arithmetic with Splitting v1	5	10
Interval Arithmetic with Splitting v2	280	580
Affine Arithmetic	555	5000
Taylor Model of 1st Order	612	750
Our approach 50 terms kept	430	600
Our approach 200 terms kept	800	1300

that could calculate bounds in a tractable time.

For range analysis, because IA suffers from dependencies of the input variables, even when this is reduced using splitting, and TwIR suffers from the need to multiply interval remainder bounds by polynomials, as we showed in Section 5.2 and equation (5.1), only AA is capable of calculating comparable bounds to our approach, and provided enough terms are kept in the polynomial, our approach can return the tightest bounds. The reason our approach can calculate tighter bounds than AA is likely to be a result of using the Handelman approach to find final bounds, which is capable of considering correlations in the rational function to improve the final bounds.

However, for relative error analysis, only our approach is able to track how relative error decreases with increasing precision, and our approach even calculates bounds close to those found by random simulation, implying our bounds are tight. This is because all the terms representing floating point error are a function of roundoff variables and input variables, meaning they are second order or greater, and hence first order methods such as AA and 1st order TwIR can only approximate these errors, whereas by retaining ‘most significant terms’, our approach can retain higher order terms.

In order to demonstrate how this can be used to improve a hardware design, we have created a basic hardware implementation which is a fully parallel implementation of the inner-most loop of Figure 5.5, where the A matrix is stored in RAM. Table 5.5 shows the resources required to create such a hardware implementation using the minimum precision necessary to ensure the relative error lies below a threshold of 5.510^{-3} according to the various methods. Using our approach, we can create a hardware design with 25% less silicon area than by using AA. The other methods cannot prove bounds, and meaning that one could either attempt a simulation based approach, which in these examples would use less hardware

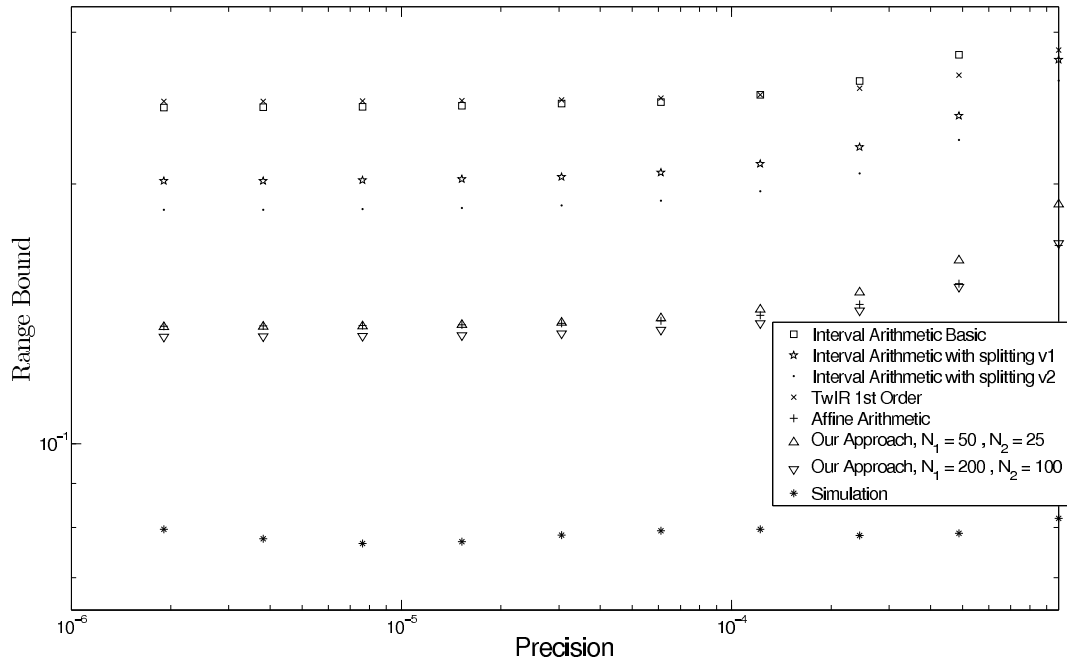
Table 5.5.: Resource usage and max frequency of 5x5 successive over relaxation.

Method	Precision (# bits)	Slices	Frequency (MHz)
Simulation	11	1252	330
Our Approach	13	1609	330
Affine Arithmetic	18	2154	300
IEEE Single Precision	23	2681	280
IEEE Double Precision	52	7983	251
IA/TwIR	∞	∞	N/A

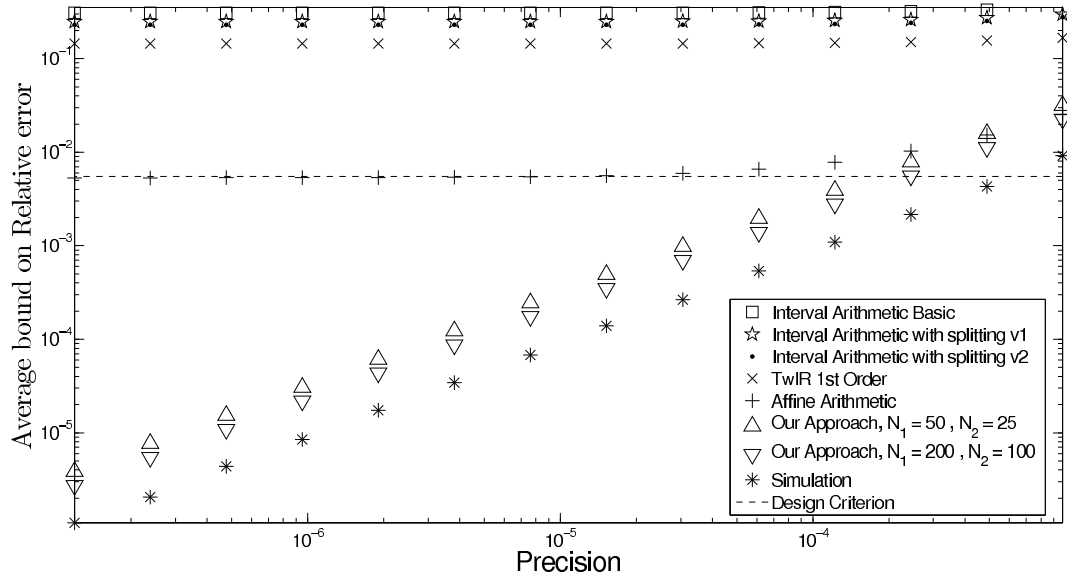
at the cost of sacrificing guarantees that it will satisfy the desired bounds, or implement it using IEEE double precision floating point units, but our approach could save up almost 80% of the silicon area in comparison. Furthermore, , because our approach can track how relative error decreases with increasing precision we could use it to tune hardware to satisfy a tighter bound on relative error, to which even AA would be unable to find a proof.

Execution time vs quality of bounds performance trade-off: It is clear that the values of N_1 and N_2 trade off tightness of bounds for execution time. The exact trade off of these will be problem dependent, however, Figure 5.9 plots this trade-off for the relative error analysis of the successive over relaxation example in an attempt to highlight the main issues. This figure shows the how bound on relative error varies for different execution times for different values of precision. If you increase the precision, as is seen from the different lines, the range gets tighter; this is expected because there is a lower amount of accumulated error. However, it is also interesting to see that if more time is spent in the algorithm, *i.e* by making N_1 and N_2 larger then we can find a tighter the bound.

In order to compare the ability of our approach to trade quality of bounds with scalability with the existing approaches, we have created Figure 5.10. In this figure, we have chosen to compare bounds on the range, because as we have shown earlier, only our approach is capable of tracking how relative error decreases with precision, and all the values in this figure are based on a mantissa of 20 bits. Furthermore, in order to compare more approaches, in Figure 5.10(a), we have restricted the successive over relaxation benchmark to only two iterations, allowing us also to see the quality of bounds and execution time trade-off for increasing the order of TwIR to 2nd order, while in Figure 5.10(b) we plot the same after the conclusion of the benchmark. For our approach, we have varied N_1 in 10 equal size increments from 20 to 400 and set $N_2 = \frac{1}{2}N_1$. Finally, to obtain a measure of the quality of the bounds,



(a) Bound on Range error.



(b) Bound on Relative error.

Figure 5.8.: Range and relative error of various methods to calculate bounds applied to a successive over relaxation of a 5x5 matrix.

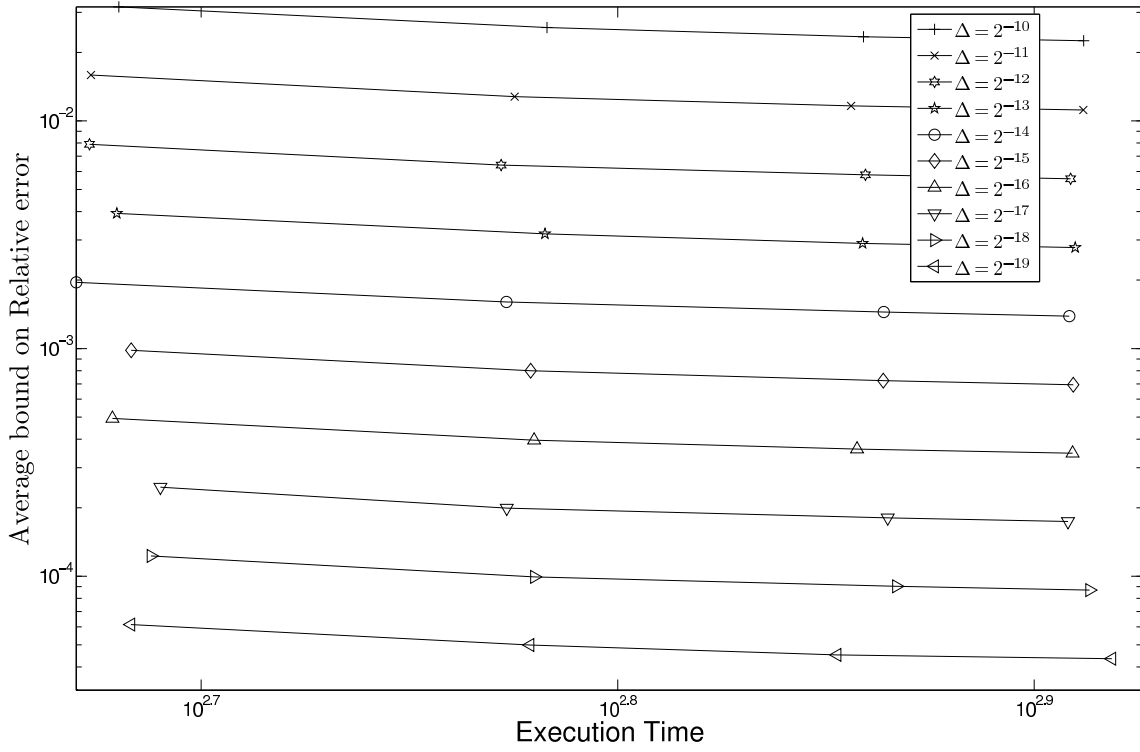
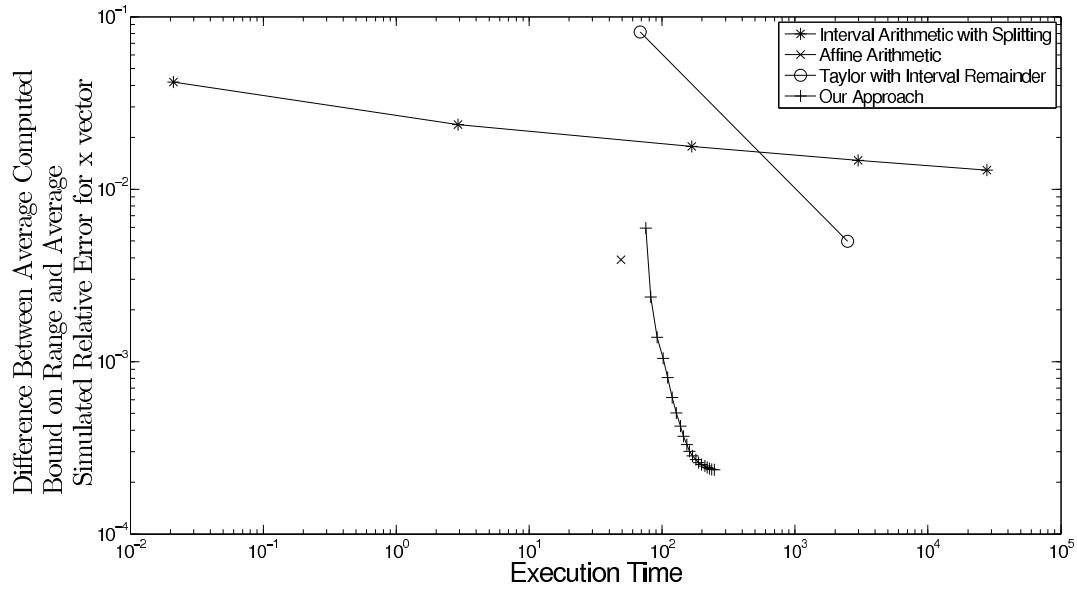
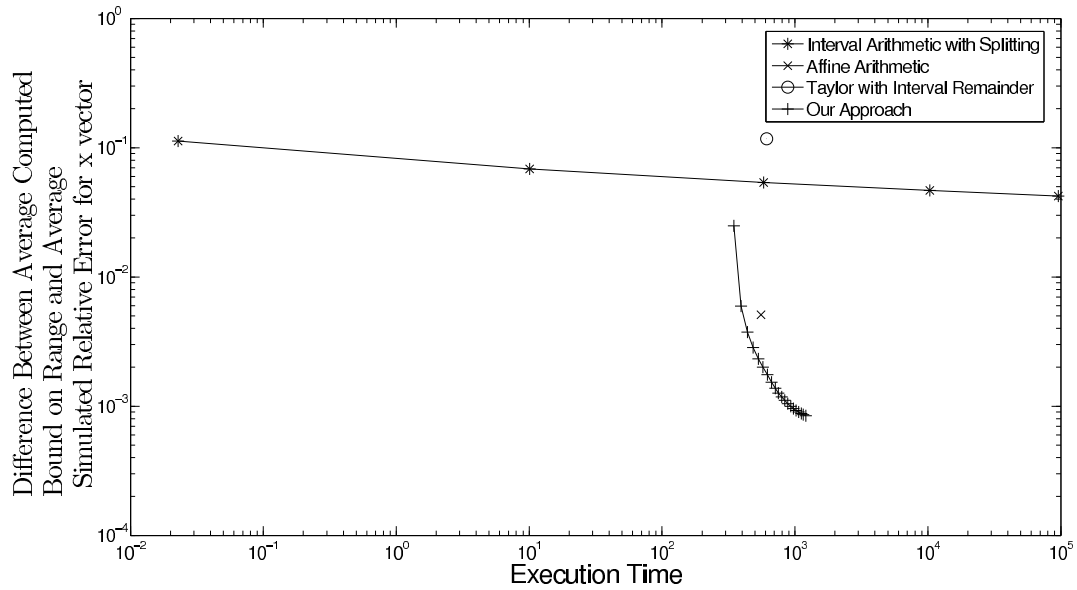


Figure 5.9.: Trade off between bound on relative error and execution time of various implementations of our approach to find the average bound on the relative error of x vector of a successive over relaxation of a 5x5 matrix.

we plotted the difference between the calculated and simulated bounds. This graph demonstrates how our method provides a much finer grain level of control in comparison to existing methods: interval splitting has a large growth in execution time for every extra split of the desired variables, AA has no control over this trade-off, and TwIR has a large difference in execution time between 1st and 2nd order. In addition, this graph shows once again that our approach is capable of computing the tightest bounds of all these methods, and because these bounds approach those found by simulation, it implies the bounds we compute are tight. While we note that in the case of Figure 5.10(a), due to the additional overheads of our algorithm in creating rational functions representing the value for intermediate variables, AA can run quicker than our approach, as the number of operations becomes larger as in Figure 5.10(a), because AA cannot trade execution time for quality of bounds, our approach has the flexibility to run quicker than AA at a cost of bounds, or for longer than AA to obtain tighter bounds.



(a) After two iterations.



(b) After seven iterations.

Figure 5.10.: Trade off between bound on range and execution time of various implementations of our approach to find the average bound on the range of x vector of a successive over relaxation of a 5x5 matrix for various approaches.

5.4.3. Test 2: MINRES

Scalability: Figure 5.11 demonstrates how the execution time of each of the methods grows with the number of operations when computing the range for intermediate variables over the course of the MINRES algorithm. We note that 2nd order TwIR once again scales poorly with the number of operations, while IA and 1st order TwIR fail to compute bounds after 228 operations because they are unable to prove the input to the $\sqrt{}$ function is non-negative. For our approach and AA, in comparison to Figures 5.7(a) and 5.7(b), in this experiment the execution time grows much faster. This is because, as mentioned in Section 5.4.1, this algorithm contains the multiplication of two polynomials or rational functions which may be large, as opposed to the multiplication of a rational functions by a single input variable. However, we comment that as the maximum number of monomials in the product of two polynomials is given by the product of the number of monomials in each polynomial and the size of the AA polynomial for intermediate variables can be much greater than $N_1 + N_2$. This ensures AA suffers much more than our approach, and is the reason our approach becomes faster than AA after much fewer operations than in Figures 5.7(a) and 5.7(b). We also comment that this graph grows in steps, unlike Figures 5.7(a) and 5.7(b), this is because operations which are products of two polynomials create many more monomials and take longer than operations which are sums of two polynomials, and hence the execution time for sums is below the worst case execution time for any operation.

Quality of bounds: Figure 5.12 shows the bounds on the average range over each of the output variables (the x -vector) for the MINRES example. In this case, only AA and our approach were capable of computing bounds. While in Figure 5.7(a), there was little difference between affine arithmetic and our approach, because of the large number of non-affine operations on polynomials, there is a noticeable difference in tightness of bounds. Though for small values of precision, our approach may fail to compute bounds, unlike AA, this could be rectified by retaining more terms, highlighting the trade-off we aim to achieve.

When attempting to calculate bounds on the relative error that would enable one to create an optimized hardware design, once again, only our approach was capable of computing bounds for this benchmark that track how relative error decreases with increasing precision, as shown in Figure 5.13, which shows the bounds on the average range over each of the output variables (the x -vector) for the MINRES exam-

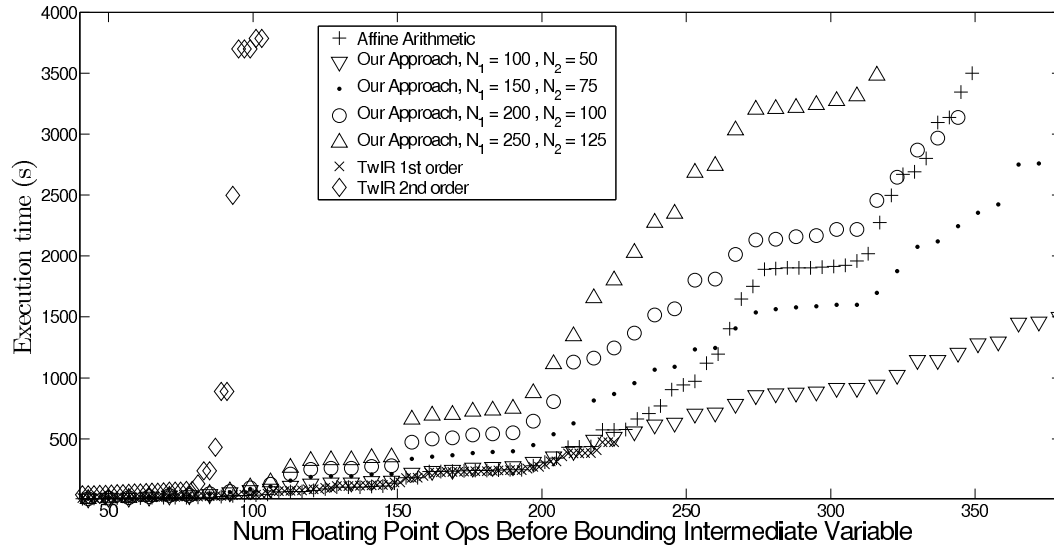


Figure 5.11.: Execution time vs number of operations using various methods to bound range applied to a MINRES algorithm of a 4x4 matrix.

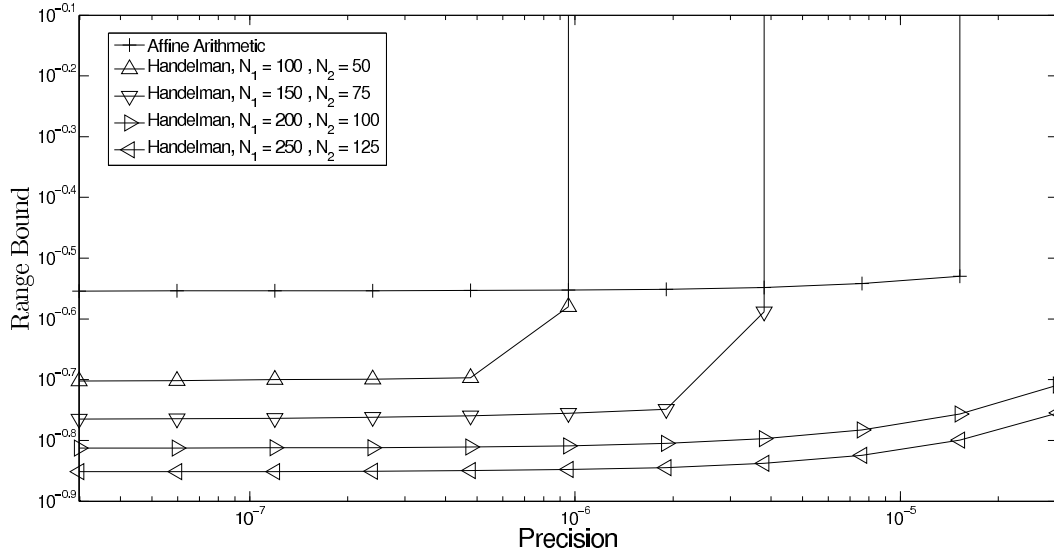


Figure 5.12.: Range bounds using various methods to bound range applied to a MINRES algorithm of a 4x4 matrix.

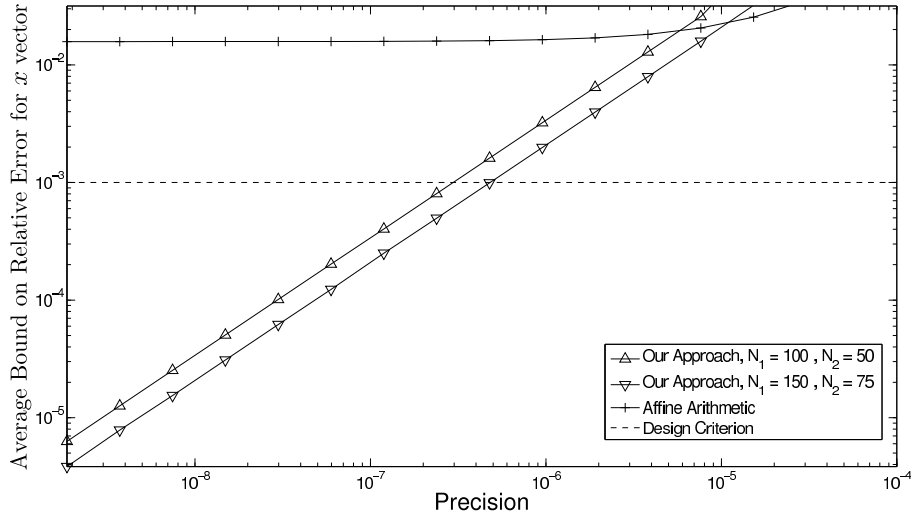


Figure 5.13.: Average bound on relative error for the x vector of the MINRES algorithm using our approach and Affine Arithmetic.

Table 5.6.: Slice use and max frequency of MINRES implementation required according to analytical tools to guarantee the relative error is less than 1×10^{-3} , or using IEEE standards.

Method	Precision (# bits)	Slices	Frequency (MHz)
Our Approach $N_1 = 150, N_2 = 75$	21	8213	225
Our Approach $N_1 = 100, N_2 = 50$	22	8371	225
AA/IA/TwIR	∞	∞	N/A
IEEE Single Precision	24	7592	150
IEEE Double Precision	52	25965	120

ple. This was similarly because errors arising from the use of finite precision arithmetic will be second order or greater, meaning first order methods can only approximate these errors. Table 5.6 shows the resource use for a parallel implementation of the MINRES algorithm required to satisfy a bound on relative error of less than 1×10^{-3} using the various methods to calculate bounds, including our approach with different run-times, as well as using IEEE single and double precision. This table again demonstrates significant savings can be made in comparison with IEEE double precision. Furthermore, by using an increased run-time, we obtain tighter bounds that result in a smaller hardware implementation, highlighting the trade-off we aim to achieve. Finally, we also note that our software has the potential to obtain a proof that IEEE single precision is sufficient to satisfy the desired precision, illustrating how our tool could also be used to obtain performance improvements on other hardware platforms.

5.5. Summary

In this chapter, we have presented a new algorithm to calculate bounds on finite precision errors that has significantly greater control over the trade-off between execution time and quality of bounds than existing methods. We have shown that the complexity of the approach we describe scales in proportion to the number of operations, unlike affine arithmetic and Taylor series with interval remainder bounds which scale in proportion to the square of the number of operations. This means that whereas the relatively small benchmarks we have used in this chapter are at the limit of to which AA and 1st order TwIR are applicable, our algorithm has the potential to compute bounds for much larger problems. Furthermore, we have incorporated this method to enhance the approach we described in Chapter 4 so that the combined approach can find tight bounds for large algorithms consisting of all elementary functions by taking into account dependencies of the intermediate variables. Finally, we once again demonstrated that these bounds can translate into smaller hardware that satisfies the same error criterion.

Chapter 6

Conclusion

This thesis has examined methods to obtain high performance in a hardware accelerator and highlighted the importance of tuning the precision used throughout any such implementation to suit a range or error specification. Furthermore, it has sought to address the difficulty of performing any such optimisation due to the lack of analytical tools which can calculate suitable bounds for the variables in an algorithm, given an input range and finite precision specification. In this chapter, we discuss the main contributions and discoveries of this thesis, before we discuss several potential strands of research that could be built upon the work we have presented in this thesis in Section 6.1.

Through the use of our initial case study, we have shown that it is possible to obtain a large performance improvement over software by using a dedicated hardware accelerator; indeed, we have shown that an overall sustained performance improvement of over an order of magnitude can be achieved by exploiting the inherent parallelism in the algorithm and using pipelining to ensure an efficient use of the operators. The potential performance of this accelerator was enhanced through the introduction of some simple new hardware structures to achieve greater flexibility in the amount of parallelism and some techniques to re-organise the RAM configurations to take advantage of certain matrix characteristics. Finally, by using integer linear programming, we have shown how to automatically optimise the use of the available hardware for a custom accelerator for a dot-product circuit or the MINRES algorithm in single precision. Altogether, these techniques obtained impressive performance figures including a large improvement over a software implementation of the MINRES algorithm. But are such improvements important? Electronic computation is becoming increasingly widespread throughout modern society,

but to continue its impact on society and use it perform more sophisticated algorithms faster, we require substantial increases in computation power. For example, suppose we had the goal of allowing electronic devices to make complex decisions intelligently, such as centrally controlling traffic flow in real-time, or predicting the effect of macro-economic choices; such a goal is far beyond modern computing power. More importantly, such performance could not be obtained by simply by process scaling due to power [86] and device constraints [138], and hence it is of increasing importance to develop design techniques to perform parallel computation efficiently to maximise the performance of any computational platform. To this end, the value of this portion of the thesis is not in the raw speed-up and performance figures, achieved for a specific algorithm, using a specific FPGA device. Instead, the contributions are the techniques described to obtain the performance improvements, for these ideas can be transferred to other hardware platforms and other algorithms. With regard to the latter point, this not only includes straightforward porting of the optimised dot-product circuit or of the technique to store symmetric matrices to another algorithm that requires it, but also using the general ideas to improve accelerators for different algorithms. Storing matrices on chip to make use of on-chip memory bandwidth and satisfy I/O constraints, maintaining a high efficiency by pipelining multiple problems into a circuit and choosing how many problems are required to fill the circuit, or using the ILP framework as a basis for a new problem are all ideas that could be used in a broader sense.

The major limitation of the techniques we described was that they failed to handle the trade-offs that exist between precision and error, which in the case of iterative algorithms affects the rate of convergence of the algorithm. However, this limitation is shared with many existing techniques aimed at creating hardware accelerators, and unlike previous work, it is a weakness that this thesis wishes to draw attention towards. In the final section of Chapter 3, Section 3.4, we explored the impact of this trade-off through simulations of how varying the precision affects the rate of convergence, and performance estimations, which interestingly could be calculated realistically by using our integer linear programming framework. This discussion helped substantiate our argument that fine tuning of the precision throughout an algorithm should not be ignored.

In an attempt to address such concerns, the rest of the thesis has turned its eye towards the field of word-length optimisation. To the mind of the author, the ultimate goal of research in the field of word-

length optimisation is to ensure algorithms are designed with some sort of error or range specification in mind, and to create a set of tools that can automatically tune the precision used throughout the hardware datapath to optimise the performance of the circuit whilst providing a guarantee that it meets the desired error specification. Over the course of this thesis, we first described how the existing techniques to optimise the precision throughout a circuit require a tool to verify that the choice of precision will satisfy the design specification. In the absence of such a tool they simulate the design, but this technique could miss corner cases meaning that the hardware could fail in practice. As a result of this study, the efforts of this thesis were placed to help create such a tool. To this end, we described a new heuristic based upon a result from real algebraic geometry that can be used to find analytical bounds for any value within an algorithm; this represents a substantial departure from the existing techniques which we have shown to all rely to some degree on interval analysis. On small computational kernels we demonstrated that because our approach can take into account dependencies within an algorithm, unlike any interval analysis based approach, it can compute much tighter bounds the existing methods. Furthermore, we also illustrated once again that this can result in superior hardware, justifying the importance of this research.

The key drawback of our heuristic is that the execution time scaled exponentially in the code size. However, we analysed the practicality of the main techniques to calculate bounds for the range or relative error of a variable within non-trivial algorithms and discovered that with the exception of a naïve implementation of interval arithmetic, the execution time of the existing methods do not scale in proportion to the number of operations. This means that for real algorithms they too will largely not be applicable. We therefore designed a set of algorithms which obtain a control over the execution time to create a scalable approach, and also provide a significant level of flexibility to trade the execution time for the potential quality of bounds for use within a word-length optimisation framework. Finally, through the use of small example algorithms, we confirmed the lack of scalability of the existing approaches that we expected from our analysis, as well the level of control and tightness of bounds that our new combined approach could achieve.

Whilst this research has shown a high degree of utility and the new heuristics that we have developed have shown performance advantages over existing techniques, they are by no means the finished article.

For example, the ideas that we have explored have helped improve both the scalability of the bounding procedure and tightness of bounds, but over the course of the thesis, we have hinted at several limitations of our current work that would benefit from further research to improve both of these aspects further. We have amalgamated a discussion of these limitations and some potential ideas for future avenues of research in the following section. However, in spite of its limitations, this thesis should have helped to make inroads into the quest to create tools to make use of the potential performance benefits available by automatically design hardware accelerators with the minimum precision necessary to meet an algorithm designers specification. More importantly, by encouraging the performance benefits of tuning the precision throughout an accelerator, describing new tools to help choose the precision required and highlighting avenues of research to develop these tools further, hopefully this thesis will give impetus to eventually change the mindset of algorithm designers such that they will always consider how much error their design can tolerate in order to exploit this potential.

6.1. Future work

6.1.1. Improving the tightness of bounds and execution time to compute bounds

While the heuristic we have proposed is capable of finding tighter bounds than many of the existing methods, it is unlikely to find the optimum bounds. This is unsurprising, given that the problem of bounding a multivariate polynomial is NP-hard, but this should not deter the search for better bounds in a tractable time. Improving the heuristic we describe in Chapter 4 is one potential avenue of research, however, a broader search would be to consider that the Handelman representation is just one special case of ‘theorems of alternatives’ in which real algebraic geometry is rich. In particular, as we discussed in Chapter 2, the proposed approach can be seen as a search over a particular form of Positivstellensatz refutation [126], an insight which could lead to further sophisticated algorithms developed in this field.

Further research could similarly be directed towards improving the algorithms from Chapter 5 which control the trade-off of execution time for quality of bounds. In this chapter, one aspect we left largely untouched in order to focus on the scalability issues was a comparison, or adaptation, of polynomial or rational approximations techniques from the field of *approximation theory* [113]; such an analysis would

be of interest to obtain tighter bounds. Alternatively, one could focus on improving the execution time of our approach by using alternative approximations for algebraic operations, for example, the work by Kinsman et al. [82] suggest relaxations for vector operations. A final potential avenue we highlighted in this chapter is the possibility of creating a hybrid optimisation strategy which combines our algorithm with a technique that performs intelligent interval splitting, such as [100], to improve bounds.

6.1.2. Handling loops and conditional statements

In the analysis of algorithms which we performed in Chapters 4 and 5, we simply unrolled the loops to create the SSA form from which find bounds for the desired variables after several iterations were computed. However, such an approach is only applicable if the loop bounds are known *a priori* at compile time, which will often not be the case. Instead an approach which can choose the internal word-lengths to ensure a loop reaches the termination conditions would be of greater interest. Moreover, if such an approach could choose internal word-lengths to reach termination within a given number of cycles, than one could analytically explore the tradeoffs between convergence, precision and performance, as we mentioned in Chapter 2. In addition, an approach that does not rely on loop unrolling is likely to have a much smaller run-time. Such an approach is likely to require a combination of work on loop termination [37] and the techniques which we have described in this thesis. Finally, it is likely that any research in this field will also be of value in choosing the word-lengths necessary to satisfy any branch conditions.

6.1.3. Automatically creating optimised hardware designs

In the examples we have used in this thesis in Chapters 4 and 5, we calculated bounds on the range or relative error for many different word-lengths, then chose the minimum precision which satisfies the chosen error or range specification. Such a procedure is simple to perform exhaustively for a small range of word-lengths, alternatively a binary search algorithm where the precision is refined according to whether the analytical tools can verify whether the chosen precision is sufficient to satisfy the design criterion would be capable of quickly finding the best global word-length for an individual design. However, the problem becomes of much greater difficulty when performing individual word-length

optimisation, as shown in [33], as the number of potential configurations grow exponentially with the number of individual operators that must be instantiated in hardware. The intuition behind individual word-length optimisation is that in many algorithms it is more important to maintain precision in some areas than others, for example, in the case of the MINRES algorithm, maintaining the precision of the matrix is important otherwise one could be converging on an incorrect matrix meaning the true error will never be minimised. As we mentioned in Chapter 2, individual word-length optimisation has been previously studied, largely in the DSP domain [35, 88], and this has led to hardware designs which use less silicon area, and hence with new techniques to calculate bounds it may be of interest to revisit some of this work targeting more general algorithms. Finally, we comment that we could also move beyond a tool which could optimise the word-lengths in a hardware design to meet an error specification to a tool that also optimises the hardware datapath to improve the overall error properties and minimise resource use, for example by taking into consideration the non-associativity of floating point operations.

6.2. Final remarks

Over the course of this thesis, we have studied the potential performance that can be obtained through tuning the precision used throughout a hardware accelerator, and worked towards creating analytical techniques which can help automatically design hardware which meets an algorithm designers specification whilst taking advantage of this performance. While this is an NP hard problem, this thesis has made inroads into this goal through developing several algorithms that are substantially different to the existing methods and can also compute tighter bounds than the existing methods, which in turn can be used to create superior hardware with guaranteed error properties. Furthermore, the contributions in this thesis should help to open several new avenues of research in this field for the community to explore in the quest to maximise the performance achievable by tuning the precision used throughout a hardware accelerator.

Appendix A

Example of our heuristic to find Handelman representations for a simple polynomial

This appendix demonstrates the use of the heuristic given in Figure 4.2 to choose the cancellation polynomials of the form (A.2) used to cancel the polynomial (A.1) taken from Chapter 4. For this example, we set $\Delta = 2^{-5}$, $x = 2^{-1}$, $y = 2^{-2}$, resulting in the polynomial given by (A.3). In this appendix, we show the chosen cancellation polynomial ($h_i(\delta)$) after each iteration of our algorithm as well as the remaining monomials in $f(\delta) + \sum_{i \in \mathbb{N}} h_i(\delta)$, and how this process eventually calculates upper bounds for the polynomial.

$$f(\delta) = -xy(xy - 1) - xy(2xy - 1)\delta_1 - x^2y^2\delta_2 - x^2y^2\delta_1^2 - 2x^2y^2\delta_1\delta_2 - x^2y^2\delta_1^2\delta_2. \quad (\text{A.1})$$

$$h(\delta) = c \prod_{j=1}^n (\Delta^{|\mu_j|} - \delta^{\mu_j})^{\alpha_j} (\Delta^{|\mu_j|} + \delta^{\mu_j})^{\beta_j}. \quad (\text{A.2})$$

:

$$f(\delta) = \frac{7}{64} + \frac{3}{32}\delta_1 - \frac{1}{64}\delta_2 - \frac{1}{64}\delta_1^2 - \frac{1}{32}\delta_1\delta_2 - \frac{1}{64}\delta_1^2\delta_2. \quad (\text{A.3})$$

Appendix A. Example of our heuristic to find Handelman representations for a simple polynomial

First iteration: $\alpha = (2, 0), \beta = (0, 1), c = 1/64$.

$$h_1(\delta) = \frac{1}{2097152} + \frac{1}{65536}\delta_2 - \frac{1}{32768}\delta_1 - \frac{1}{1024}\delta_1\delta_2 + \frac{1}{2048}\delta_1^2 + \frac{1}{64}\delta_1^2\delta_2 \quad (\text{A.4})$$

$$f(\delta) + \sum_{i=1}^1 h_i(\delta) = \frac{229377}{2097152} + \frac{3071}{32768}\delta_1 - \frac{1023}{65536}\delta_2 - \frac{31}{2048}\delta_1^2 - \frac{33}{1024}\delta_1\delta_2 \quad (\text{A.5})$$

Second iteration: $\alpha = (1, 1), \beta = (0, 0), c = 33/1024$.

$$h_2(\delta) = \frac{33}{1048576} - \frac{33}{32768}\delta_2 - \frac{33}{32768}\delta_1 + \frac{33}{1024}\delta_1\delta_2 \quad (\text{A.6})$$

$$f(\delta) + \sum_{i=1}^2 h_i(\delta) = \frac{229443}{2097152} + \frac{1519}{16384}\delta_1 - \frac{1089}{65536}\delta_2 - \frac{31}{2048}\delta_1^2 \quad (\text{A.7})$$

Third iteration: $\alpha = (2, 0), \beta = (0, 0), c = 31/2048$.

$$h_3(\delta) = \frac{31}{2097152} - \frac{31}{32768}\delta_1 + \frac{31}{2048}\delta_1^2 \quad (\text{A.8})$$

$$f(\delta) + \sum_{i=1}^3 h_i(\delta) = \frac{114737}{1048576} + \frac{3007}{32768}\delta_1 - \frac{1089}{65536}\delta_2 \quad (\text{A.9})$$

Forth iteration: $\alpha = (0, 0), \beta = (0, 1), c = 1089/65536$.

$$h_4(\delta) = \frac{1089}{2097152} + \frac{1089}{65536}\delta_2 \quad (\text{A.10})$$

$$f(\delta) + \sum_{i=1}^4 h_i(\delta) = \frac{230563}{2097152} + \frac{3007}{32768}\delta_1; \quad (\text{A.11})$$

Fifth iteration: $\alpha = (0, 0), \beta = (1, 0), c = 3007/32768$.

$$h_5 = \frac{3007}{1048576} - \frac{3007}{32768}\delta_1 \quad (\text{A.12})$$

$$f(\delta) + \sum_{i=1}^5 h_i(\delta) = \frac{236577}{2097152} \quad (\text{A.13})$$

After this stage, we note this could be written as $\frac{236577}{2097152} - f(\delta) = \sum_{i=1}^5 h_i(\delta)$. As a result, $\sum_{i=1}^5 h_i(\delta)$ is a Generalised Handelman Representation (as described in Chapter 4) for $\frac{236577}{2097152} - f(\delta)$

Appendix A. Example of our heuristic to find Handelman representations for a simple polynomial

, meaning $\frac{236577}{2097152} - f(\delta)$ is non-negative, and hence $\frac{236577}{2097152} \geq f(\delta)$. We note that these bounds are tighter than those bounds found using interval arithmetic on the polynomial (A.1), as shown in (A.14), which corresponds to the approach in Chapter 4 that we described as ‘interval arithmetic on the simplified polynomial’.

$$\begin{aligned}
 7/64 &\in [7/64; 7/64] \\
 +\frac{3/32}{\delta_1} &\in [-3/1024; 3/1024] \\
 +\frac{1/64}{\delta_2} &\in [-1/2048; 1/2048] \\
 +\frac{1/64^2}{\delta_1} &\in [-1/65536; 1/65536] \\
 +\frac{1/32}{\delta_1} \delta_2 &\in [-1/32768; 1/32768] \\
 +\frac{1/64^2}{\delta_1} \delta_2 &\in [-1/2097152; 1/2097152] \\
 f(\delta) &\in [-236641/2097152; 236641/2097152] \tag{A.14}
 \end{aligned}$$

Bibliography

- [1] ABEL, U. On the Lagrange remainder of the Taylor formula. *The American Mathematical Monthly* 110, 7 (2003), pp. 627–633.
- [2] ADAMS, W. W., AND LOUSTAUNAU., P. *An Introduction to Grobner Bases*. American Mathematical Society, 1994.
- [3] ALAM, S., AGARWAL, P., SMITH, M., VETTER, J., AND CALIGA, D. Using FPGA devices to accelerate biomolecular simulations. *Computer* 40, 3 (march 2007), 66 –73.
- [4] ALTERA. White paper, Stratix iii FPGAs vs. Xilinx Virtex-5 devices: Architecture and performance comparison. Tech. rep., October 2007.
- [5] BARRETT, R., BERRY, M., CHAN, T. F., DEMMEL, J., DONATO, J., DONGARRA, J., EIKHOUT, V., POZO, R., ROMINE, C., AND DER VORST, H. V. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
- [6] BAZARAA, M. S. *Nonlinear programming : theory and algorithms*. Wiley, 2006.
- [7] BELANOVIC, P., AND LEESER, M. A library of parameterized floating-point modules and their use. In *Int. Proc. of the Reconfigurable Computing Is Going Mainstream, Conf. on Field-Programmable Logic and Applications* (London, UK, 2002), Springer-Verlag, pp. 657–666.
- [8] BERZ, M., HOFFSTTTER, G., AND HOFFSTÄTTER, G. Computation and application of taylor polynomials with interval remainder bounds. *Reliable Computing* 4 (1998), 83–97.

Bibliography

- [9] BIGLIERI, E., CALDERBANK, R., CONSTANTINIDES, A., GOLDSMITH, A., PAULRAJ, A., AND POOR, H. V. *MIMO Wireless Communications*. Cambridge University Press, 2007.
- [10] BISHOP, D. VHDL-2008 support library. <http://www.vhdl.org/fphdl/>, 2008.
- [11] BOLAND, D., AND CONSTANTINIDES, G. An FPGA-based implementation of the MINRES algorithm. In *Proc. Int. Conf. Field Programmable Logic and Applications* (Sept. 2008), pp. 379–384.
- [12] BOLAND, D., AND CONSTANTINIDES, G. Optimising memory bandwidth use for matrix-vector multiplication in iterative methods. In *Proc. Int. Symp. Applied Reconfigurable Computing* (2010), vol. 5992, pp. 169–181.
- [13] BOLAND, D., AND CONSTANTINIDES, G. Bounding variable values and round-off effects using handelman representations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 11 (Nov. 2011), 1691–1704.
- [14] BOLAND, D., AND CONSTANTINIDES, G. Optimising memory bandwidth use and performance for matrix-vector multiplication in iterative methods. *ACM Trans. Reconfigurable Technol. Syst.* 4 (August 2011), 22:1–22:14.
- [15] BOLAND, D., AND CONSTANTINIDES, G. A scalable approach for automated precision analysis. *accepted to appear in Proc. Int. Symp. on Field Programmable Gate Arrays* (2012).
- [16] BOLAND, D., AND CONSTANTINIDES, G. A. Automated precision analysis: A polynomial algebraic approach. In *Proc. Int. Symp. Field-Programmable Custom Computing Machines* (Los Alamitos, CA, USA, 2010), IEEE Computer Society, pp. 157–164.
- [17] BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *ACM SIGGRAPH Papers* (New York, NY, USA, 2003), ACM, pp. 917–924.

- [18] BONDALAPATI, K., AND PRASANNA, V. Dynamic precision management for loop computations on reconfigurable architectures. In *Proc. Int. Symp. Field-Programmable Custom Computing Machines* (1999), pp. 249–258.
- [19] BOYD, S., AND VANDENBERGHE, L. *Convex Optimization*. Cambridge University Press, March 2004.
- [20] BROWN, S., AND ROSE, J. Architecture of FPGAs and CPLDs: A tutorial. *IEEE Transactions on Design and Test of Computers* 13 (1996), 42–57.
- [21] BUTTARI, A., DONGARRA, J., KURZAK, J., LANGOU, J., LUSZCZEK, P., AND TOMOV, S. The impact of multicore on math software. In *Proc. Int. Conf. on Applied parallel computing: state of the art in scientific computing* (Berlin, Heidelberg, 2007), PARA’06, Springer-Verlag, pp. 1–10.
- [22] BUTTARI, A., DONGARRA, J., KURZAK, J., LUSZCZEK, P., AND TOMOV, S. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.* 34, 4 (2008), 1–22.
- [23] CALLANAN, O., GREGG, D., NISBET, A., AND PEARDON, M. High performance scientific computing using FPGAs with IEEE floating point and logarithmic arithmetic for lattice QCD. *Proc. Int. Conf. Field Programmable Logic and Applications* (Aug. 2006), 1–6.
- [24] CALVETTI, D., REICHEL, L., AND SORENSEN, D. C. An implicitly restarted Lanczos method for large symmetric eigenvalue problems. *Electron. Trans. Numerical. Anal.* 2 (1994), 1–21.
- [25] CANTIN, M.-A., SAVARIA, Y., AND LAVOIE, P. A comparison of automatic word length optimization procedures. In *Proc. Int. Symp. on Circuits and Systems* (2002), vol. 2, pp. II–612 – II–615 vol.2.
- [26] CANTIN, M.-A., SAVARIA, Y., PRODANOS, D., AND LAVOIE, P. An automatic word length determination method. In *Proc. Int. Symp. on Circuits and Systems* (2001), vol. 5, pp. 53–56 vol. 5.

- [27] CHANG, M. L., AND HAUCK, S. Précis: A design-time precision analysis tool. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 229–238.
- [28] CHANG, M. L., AND HAUCK, S. Automated least-significant bit datapath optimization for FPGAs. *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines 0* (2004), 59–67.
- [29] CHOW, G., KWOK, K., LUK, W., AND LEONG, P. Mixed precision processing in reconfigurable systems. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines* (may 2011), pp. 17 –24.
- [30] CMAR, R., RIJNDERS, L., SCHAUMONT, P., VERNALDE, S., AND BOLSENS, I. A methodology and design environment for DSP ASIC fixed point refinement. In *Proceedings of the conference on design, automation and test in Europe* (New York, NY, USA, 1999), DATE '99, ACM.
- [31] CONG, J., GURURAJ, K., LIU, B., LIU, C., ZHANG, Z., ZHOU, S., AND ZOU, Y. Evaluation of static analysis techniques for fixed-point precision optimization. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2009), FCCM '09, IEEE Computer Society, pp. 231–234.
- [32] CONSTANTINIDES, G. Perturbation analysis for word-length optimization. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines* (april 2003), pp. 81 – 90.
- [33] CONSTANTINIDES, G., CHEUNG, P., AND LUK, W. The multiple wordlength paradigm. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines* (29 2001-april 2 2001), pp. 51 –60.
- [34] CONSTANTINIDES, G., CHEUNG, P., AND LUK, W. Optimum wordlength allocation. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines* (2002), pp. 219–228.

- [35] CONSTANTINIDES, G., CHEUNG, P., AND LUK, W. Wordlength optimization for linear digital signal processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22, 10 (2003), 1432–1442.
- [36] CONSTANTINIDES, G. A., CHEUNG, P. Y. K., AND LUK, W. *Synthesis And Optimization Of DSP Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.
- [37] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 415–426.
- [38] COURTOIS, N., KLIMOV, A., PATARIN, J., AND SHAMIR, A. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Proc. Int. Conf. on Theory and application of cryptographic techniques* (Berlin, Heidelberg, 2000), Springer-Verlag, pp. 392–407.
- [39] DAI, Y., AND YUAN, Y. Convergence properties of the Beale-Powell restart algorithm, 1998.
- [40] DAUMAS, M., AND MELQUIOND, G. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.* 37 (January 2010), 2:1–2:20.
- [41] DAUMAS, M., MELQUIOND, G., AND MUNOZ, C. Guaranteed proofs using interval arithmetic. In *Proc. Int. Symp. on Computer Arithmetic 2005. ARITH-17 2005*. (june 2005), pp. 188 – 195.
- [42] DE DINECHIN, F., DETREY, J., CRET, O., AND TUDORAN, R. When FPGAs are better at floating-point than microprocessors. In *Proc. Int. Symp. Field programmable gate arrays* (2008), p. 260.
- [43] DE DINECHIN, F., LAUTER, C., AND MELQUIOND, G. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Transactions on Computers* 60 (2011), 242–253.

Bibliography

- [44] DE DINECHIN, F., PASCA, B., CRET, O., AND TUDORAN, R. An FPGA-specific approach to floating-point accumulation and sum-of-products. In *Proc. Int. Conf. on ICECE Technology, 2008. FPT 2008*. (2008), pp. 33–40.
- [45] DE FIGUEIREDO, L. H., AND STOLFI, J. *Self-Validated Numerical Methods and Applications*. Brazilian Mathematics Colloquium monographs. IMPA/CNPq, Rio de Janeiro, Brazil, 1997.
- [46] DE MATOS, G., AND NETO, H. On reconfigurable architectures for efficient matrix inversion. *Proc. Int. Conf. Field Programmable Logic and Applications* (Aug. 2006), 369–374.
- [47] DELORIMIER, M., AND DEHON, A. Floating-point sparse matrix-vector multiply for FPGAs. In *Proc. Int. Symp. on Field-Programmable Gate Arrays* (New York, NY, USA, 2005), ACM, pp. 75–85.
- [48] DONGARRA, J. J. Performance of various computers using standard linear equations software.
- [49] EINARSSON, B. *Handbook on Accuracy and Reliability in Scientific Computation*. Soc for Industrial & Applied Math, 2005, ch. 10, pp. 195–240.
- [50] EL-KURDI, Y., GROSS, W. J., AND GIANNACOPOULOS, D. Sparse matrix-vector multiplication for finite element method matrices on FPGAs. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 293–294.
- [51] ELKURDI, Y., FERNNDEZ, D., SOULEIMANOV, E., GIANNACOPOULOS, D., AND GROSS, W. J. FPGA architecture and implementation of sparse matrix-vector multiplication for the finite element method. *Computer Physics Communications* 178, 8 (2008), 558 – 570.
- [52] FANG, C. F., RUTENBAR, R. A., AND CHEN, T. Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs. In *Proc. Int. Conf. on Computer-aided design* (Washington, DC, USA, 2003), ICCAD '03, IEEE Computer Society, pp. 275–.

- [53] FANG, C. F., RUTENBAR, R. A., PÜSCHEL, M., AND CHEN, T. Toward efficient static analysis of finite-precision effects in dsp applications via affine arithmetic modeling. In *Proc. Int. Design Automation Conference* (New York, NY, USA, 2003), DAC '03, ACM, pp. 496–501.
- [54] FARIN, G. *Curves and surfaces for CAGD: a practical guide*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [55] FISHER, B. *Polynomial Based Iteration Methods for Symmetric Linear Systems*. Wiley, Teubner, Baltimore, MD, USA, 1996.
- [56] FOUSSE, L., HANROT, G., LEFÈVRE, V., PÉLISSIER, P., AND ZIMMERMANN, P. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.* 33, 2 (June 2007).
- [57] GAFFAR, A. A., LUK, W., CHEUNG, P. Y. K., SHIRAZI, N., AND HWANG, J. Automating customisation of floating-point designs. In *Proc. Int. Conf. on Reconfigurable Computing Is Going Mainstream, Field-Programmable Logic and Applications* (London, UK, UK, 2002), FPL '02, Springer-Verlag, pp. 523–533.
- [58] GARLOFF, J., JANSSON, C., AND SMITH, A. P. Lower bound functions for polynomials. *J. Comput. Appl. Math.* 157 (August 2003), 207–225.
- [59] GARLOFF, J., AND SMITH, A. P. Rigorous affine lower bound functions for multivariate polynomials and their use in global optimisation. *Proc. Int. Conf. Applied Operational Research 1* (2008), 199–211.
- [60] GEDDES, K. O., AND ZHENG, W. W. Exploiting fast hardware floating point in high precision computation. In *Proc. Int. Symp. on Symbolic and algebraic computation* (New York, NY, USA, 2003), ISSAC '03, ACM, pp. 111–118.
- [61] GÖDDEKE, D., STRZODKA, R., AND TUREK, S. Accelerating double precision FEM simulations with GPUs. In *Proc. Symp. on Simulation Technique* (Sep 2005).

Bibliography

- [62] GOLUB, G. H., AND LOAN, C. F. V. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.
- [63] GOVINDU, G., SCROFANO, R., AND PRASANNA, V. K. A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. *Proc. Int. Conf. Engineering Reconfigurable Systems and Algorithms* (2005), 137–148.
- [64] GREENBAUM, A. *Iterative methods for solving linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.
- [65] HALFHILL, T. Parallel Processing with CUDA. *Microprocessor Journal* (January 2008).
- [66] HAMMER, R., RATZ, D., KULISCH, U., AND HOCKS, M. *C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.
- [67] HANDELMAN, D. Representing polynomials by positive linear functions on compact convex polyhedra. *Pac. J. Math* 132, 1 (1988), 35–62.
- [68] HANSEN, E. A generalized interval arithmetic. In *Interval Mathematics*, K. Nickel, Ed., vol. 29 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1975, pp. 7–18.
- [69] HANSEN, E., AND WALSTER, G. W. *Global optimization using interval analysis*, vol. 264 of *Monographs and Textbooks in Pure and Applied Mathematics*. Marcel Dekker Inc., New York, 2004.
- [70] HEATH, M. T. *Scientific Computing*. McGraw-Hill Higher Education, 2001.
- [71] HERDE, C., EGGERS, A., FRANZLE, M., AND TEIGE, T. Analysis of hybrid systems using hysat. In *Int. Conf. on Systems* (2008), pp. 196 –201.
- [72] HESTENES, M. R., AND STIEFEL, E. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards* 49 (Dec 1952), 409–436.

- [73] HIGHAM, N. J. *Accuracy and Stability of Numerical Algorithms*, second ed. Soc for Industrial & Applied Math, Philadelphia, PA, USA, 2002.
- [74] HOEKSTRA, A. G., SLOOT, P., HOFFMANN, W., AND HERTZBERGER, L. Time complexity of a parallel conjugate gradient solver for light scattering simulations: Theory and spmd implementation. Tech. rep., 1992.
- [75] HOLLAND, B., NAGARAJAN, K., CONGER, C., JACOBS, A., AND GEORGE, A. D. RAT: a methodology for predicting performance in application design migration to FPGAs. In *Proc. Workshop on High-performance reconfigurable computing technology and applications* (New York, NY, USA, 2007), ACM, pp. 1–10.
- [76] HU, J., QUIGLEY, S., AND CHAN, A. An element-by-element preconditioned conjugate gradient solver of 3D tetrahedral finite elements on an FPGA coprocessor. *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on* (Sept. 2008), 575–578.
- [77] IEEE. *IEEE Std 754-1985 for Binary Floating-Point Arithmetic*. 1985.
- [78] IEEE. *IEEE Std 754-2008 for Binary Floating-Point Arithmetic*. 2008.
- [79] ILOG, INC. Solver cplex, 2009. <http://www.ilog.fr/products/cplex/> (accessed 02 November 2009).
- [80] KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JOHNS, C. R., MAEURER, T. R., AND SHIPPY, D. Introduction to the cell multiprocessor. *IBM J. Res. Dev.* 49, 4/5 (2005), 589–604.
- [81] KEDING, H., WILLEMS, M., COORS, M., AND MEYR, H. Fridge: a fixed-point design and simulation environment. In *Proc. Conf. on Design, Automation and Test in Europe* (Washington, DC, USA, 1998), DATE '98, IEEE Computer Society, pp. 429–435.
- [82] KINSMAN, A., AND NICOLICI, N. Computational bit-width allocation for operations in vector calculus. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on* (oct. 2009), pp. 433–438.

- [83] KINSMAN, A. B., AND NICOLICI, N. Finite precision bit-width allocation using SAT-modulo theory. In *Proc. Conf. on Design, Automation and Test in Europe* (2009).
- [84] KINSMAN, A. B., AND NICOLICI, N. Bit-width allocation for hardware accelerators for scientific computing using sat-modulo theory. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 29 (March 2010), 405–413.
- [85] KINSMAN, A. B., AND NICOLICI, N. Robust design methods for hardware accelerators for iterative algorithms in scientific computing. In *Proc. Design Automation Conference* (2010), pp. 254–257.
- [86] KOOMEY, J. G. Estimating total power consumption by servers in the U.S. and the world. Tech. rep., Lawrence Berkley National Laboratory, Feb 2007.
- [87] KUM, K.-I., AND SUNG, W. Word-length optimization for high-level synthesis of digital signal processing systems. In *Signal Processing Systems, 1998. SIPS 98. 1998 IEEE Workshop on* (oct 1998), pp. 569–578.
- [88] KUM, K.-I., AND SUNG, W. Combined word-length optimization and high-level synthesis of digital signal processing systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 8 (Aug 2001), 921–930.
- [89] LANGHAMMER, M. Floating point datapath synthesis for FPGAs. *Int. Conf. on Field Programmable Logic and Applications* (Sept. 2008), 355–360.
- [90] LANGOU, J., LUSZCZEK, P., KURZAK, J., BUTTARI, A., AND DONGARRA, J. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *Proc. Supercomputing Conference* (nov. 2006), pp. 50–67.
- [91] LASSERRE, J. B. Polynomial programming: LP-relaxations also converge. *SIAM J. on Optimization* 15, 2 (2005), 383–393.

Bibliography

- [92] LEE, D.-U., GAFFAR, A., CHEUNG, R., MENCER, O., LUK, W., AND CONSTANTINIDES, G. Accuracy-guaranteed bit-width optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 10 (Oct. 2006), 1990–2000.
- [93] LEE, D.-U., GAFFAR, A. A., MENCER, O., AND LUK, W. Minibit: bit-width optimization via affine arithmetic. In *Proc. Design Automation Conference* (New York, NY, USA, 2005), ACM, pp. 837–840.
- [94] LINDERMAN, M. D., HO, M., DILL, D. L., MENG, T. H., AND NOLAN, G. P. Towards program optimization through automated analysis of numerical precision. In *Proc. Int. Symp. on Code generation and optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 230–237.
- [95] LOPES, A., AND CONSTANTINIDES, G. A high throughput FPGA-based floating point conjugate gradient implementation for dense matrices. *to appear in ACM Transactions on Reconfigurable Technology and Systems* (2008).
- [96] LOPES, A., CONSTANTINIDES, G., AND KERRIGAN, E. A floating-point solver for band structured linear equations. In *Int. Conf. on Field Programmable Technology. ICECE 2008* (dec. 2008), pp. 353–356.
- [97] LOPES, A. R., AND CONSTANTINIDES, G. A. A high throughput FPGA-based floating point conjugate gradient implementation. In *Proc. Applied Reconfigurable computing* (2008), pp. 75–86.
- [98] LOPES, A. R., AND CONSTANTINIDES, G. A. A fused hybrid floating-point and fixed-point dot-product for FPGAs. In *Proc. Int. Symp. on Applied Reconfigurable Computing* (2010), pp. 157–168.
- [99] LOPES, A. R., SHAHZAD, A., CONSTANTINIDES, G., AND KERRIGAN, E. More FLOPS or more precision? Accuracy parameterizable linear equations solvers for model-predictive control. In *Proc. Int. Symp. on Field-Programmable Custom Computing Machines* (2009).

- [100] LÖTHI, J., AND M. LLADÓ, C. Splitting techniques for interval parameters and their application to performance models. *Performance Evaluation* 51, 1 (2003), 47 – 74.
- [101] LUNDGREN, D. FPU double VHDL. http://opencores.org/project,fpu_double.
- [102] MACIEJOWSKI, J. M. *Predictive control with constraints*. Prentice Hall, Essex, England, 2002.
- [103] MAKINO, K., AND BERZ, M. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics* 4 (2003), 379–456.
- [104] MARTIN, R., SHOU, H., VOICULESCU, I., BOWYER, A., AND WANG, G. Comparison of interval methods for plotting algebraic curves. *Comput. Aided Geom. Des.* 19 (July 2002), 553–587.
- [105] MASLENNIKOW, O., LEPEKHA, V., AND SERGYIENKO, A. FPGA implementation of the conjugate gradient method. In *Proc. of the Parallel Processing and Applied Mathematics* (2005), R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, Eds., vol. 3911 of *Lecture Notes in Computer Science*, Springer, pp. 526–533.
- [106] MEURANT, G., AND STRAKOS, Z. The Lanczos and conjugate gradient algorithms in finite precision arithmetic. *Acta Numerica* 15 (2006), 471–542.
- [107] MITTELMANN, H. An independent benchmarking of sdp and socp solvers. *Mathematical Programming* 95 (2003), 407–430. 10.1007/s10107-002-0355-5.
- [108] MOORE, R. E. *Interval Analysis*. Prentice-Hall, Englewood Cliff, NJ, 1966.
- [109] MORRIS, G. R., AND PRASANNA, V. K. A pipelined-loop-compatible architecture and algorithm to reduce variable-length sets of floating-point data on a reconfigurable computer. *J. Parallel Distrib. Comput.* 68, 7 (2008), 913–921.
- [110] MORRIS, G. R., PRASANNA, V. K., AND ANDERSON, R. D. An FPGA-based application-specific processor for efficient reduction of multiple variable-length floating-point data sets. In *Proc. Int. Conf. on Application-specific Systems, Architectures and Processors* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 323–330.

Bibliography

- [111] MORRIS, G. R., PRASANNA, V. K., AND ANDERSON, R. D. A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer. In *Proc. Int. Symp. Field-Programmable Custom Computing Machines* (2006), pp. 3–12.
- [112] MORTON, K. W., AND MAYERS, D. F. *Numerical Solution of Partial Differential Equations: An Introduction*. Cambridge University Press, New York, NY, USA, 2005.
- [113] MULLER, J.-M. *Elementary Functions: Algorithms and Implementation*. Birkhauser, Boston, USA, 2006.
- [114] NAYAK, A., HALDAR, M., CHOUDHARY, A., AND BANERJEE, P. Precision and error analysis of matlab applications during automated hardware synthesis for FPGAs. In *Proc. conf. on Design, Automation and Test in Europe* (Piscataway, NJ, USA, 2001), DATE '01, IEEE Press, pp. 722–728.
- [115] NEUMAIER, A. Taylor Forms Use and limits. *Reliable Computing* 9 (2003), 43–79. 10.1023/A:1023061927787.
- [116] NOCEDAL, J., AND WRIGHT, S. J. *Numerical Optimization*. Springer, New York, USA, 1999.
- [117] NVIDIA. Tesla C1060 computing processor board. Tech. rep., January 2007.
- [118] OSBORNE, W., CHEUNG, R., COUTINHO, J., LUK, W., AND MENCER, O. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. In *Int. Conf. on Field Programmable Logic and Applications* (Aug. 2007), pp. 617–620.
- [119] OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRGER, J., LEFOHN, A. E., AND PURCELL, T. J. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 26, 1 (2007), 80–113.
- [120] OWRE, S., RUSHBY, J. M., , AND SHANKAR, N. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)* (Saratoga, NY, jun 1992), D. Kapur, Ed., vol. 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 748–752.

- [121] PAIGE., C. C. *The computation of eigenvalues and eigenvectors of very large sparse matrices*. PhD thesis, University of London, London, England, 1971.
- [122] PAIGE, C. C., AND SAUNDERS, M. A. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis* 12, 4 (Sept 1975), 617–629.
- [123] PAL, L., AND CSENDES, T. Intlab implementation of an interval global optimization algorithm. *Optimization Methods Software* 24 (August 2009), 749–759.
- [124] PANG, Y., RADECKA, K., AND ZILIC, Z. Optimization of imprecise circuits represented by taylor series and real-valued polynomials. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 29 (August 2010), 1177–1190.
- [125] PANG, Y., RADECKA, K., AND ZILIC, Z. An efficient hybrid engine to perform range analysis and allocate integer bit-widths for arithmetic circuits. In *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific* (jan. 2011), pp. 455 –460.
- [126] PARRILO, P. A. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming Ser. B* 96, 2-3 (2003), 293–320.
- [127] PRIEST, D. Algorithms for arbitrary precision floating point arithmetic. In *Proc. IEEE. Symp. Computer Arithmetic* (June 1991), pp. 132–143.
- [128] RADECKA, K., AND ZILIC, Z. Arithmetic transforms for compositions of sequential and imprecise datapaths. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 25, 7 (july 2006), 1382 –1391.
- [129] RALL, L. B. Automatic differentiation: Techniques and applications. *Lecture Notes in Computer Science*.
- [130] RATSCHKE, H. Centered forms. *SIAM Journal on Numerical Analysis* 17, 5 (1980), pp. 656–662.
- [131] RUMP, S. INTLAB - INTerval LABoratory. In *Developments in Reliable Computing*, T. Csendes, Ed. Kluwer Academic Publishers, Dordrecht, 1999, pp. 77–104. <http://www.ti3.tu-harburg.de/rump/>.

Bibliography

- [132] RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., STONE, S. S., KIRK, D. B., AND HWU, W.-M. W. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proc. Symp. on Principles and practice of parallel programming* (New York, NY, USA, 2008), PPOPP '08, ACM, pp. 73–82.
- [133] SAAD, Y. *Iterative Methods for Sparse Linear Systems, 1st Edition*. PWS, April 1996.
- [134] SEVERANCE, C. IEEE 754: An interview with William Kahan. *Computer* 31, 3 (mar 1998), 114–115.
- [135] SEWELL, G. *The numerical solution of ordinary and partial differential equations*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.
- [136] SMITH, A. P. Fast construction of constant bound functions for sparse polynomials. *J. of Global Optimization* 43 (March 2009), 445–458.
- [137] STEPHENSON, M., BABB, J., AND AMARASINGHE, S. Bitwidth analysis with application to silicon compilation. In *Proc. conf. on Programming Language Design and Implementation* (2000), pp. 108–120.
- [138] STOTT, E., SEDCOLE, P., AND CHEUNG, P. Fault tolerance and reliability in field-programmable gate arrays. *IET Computers Digital Techniques* 4, 3 (may 2010), 196–210.
- [139] STRZODKA, R., AND GÖDDEKE, D. A. Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In *IEEE Proceedings on Field-Programmable Custom Computing Machines (FCCM 2006)* (May 2006), IEEE Computer Society Press, pp. 259–268. doi: 10.1109/FCCM.2006.57.
- [140] SUN, J., PETERSON, G., AND STORAASLI, O. Sparse matrix-vector multiplication design on FPGAs. In *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 349–352.

Bibliography

- [141] SUN, J., PETERSON, G., AND STORAASLI, O. High-performance mixed-precision linear solver for FPGAs. *Computers, IEEE Transactions on* 57, 12 (dec. 2008), 1614–1623.
- [142] SUNG, W., AND KUM, K.-I. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *Signal Processing, IEEE Transactions on* 43, 12 (dec 1995), 3087–3090.
- [143] SUTTER, H. A fundamental turn toward concurrency in software. *Dr. Dobbs Journal* (March 2005).
- [144] THE COQ DEVELOPMENT TEAM. *The Coq Proof Assistant Reference Manual – Version V7.1*, Oct. 2001. <http://coq.inria.fr>.
- [145] TURKINGTON, K., MASSELOS, K., CONSTANTINIDES, G., AND LEONG, P. FPGA based acceleration of the LINPACK benchmark: A high level code transformation approach. *Proc. Int. Conf. Field Programmable Logic and Applications* (Aug. 2006), 375–380.
- [146] UNDERWOOD, K. FPGAs vs. CPUs: trends in peak floating-point performance. In *Proc. Int. Symp. Field Programmable Gate Arrays* (2004), pp. 171–180.
- [147] V., L. K., M., M. J., AND F., W. B. Multiplexed model predictive control. Technical report, Cambridge University Engineering Dept, July 2006. CUED/F-INFENG/TR.561.
- [148] VORST, H. A. V. D. *Iterative methods for large linear systems*. Lecture notes on iterative methods,, 2002.
- [149] WANG, X., BRAGANZA, S., AND LEESER, M. Advanced components in the variable precision floating-point library. *Proc. Int. Symp. on Field-Programmable Custom Computing Machines* (April 2006), 249–258.
- [150] WANG, X., AND LEESER, M. Variable precision floating point division and square root. *Workshop on High Performance Embedded Computing* (2004), 47–48.

- [151] WANG, X., AND LEESER, M. Efficient FPGA implementation of QR decomposition using a systolic array architecture. In *Proc. 16th Int. Symp. Field Programmable Gate Arrays* (2008), p. 260.
- [152] WILLEMS, M., BURSGENS, V., KEDING, H., GROTKER, T., AND MEYR, H. System level fixed-point design based on an interpolative approach. In *Proc. Design Automation Conference* (jun 1997), pp. 293–298.
- [153] WINSTON, W. L. *Introduction to Mathematical Programming: Applications and Algorithms*. Duxbury Resource Center, 2003.
- [154] WU, K., CANNING, A., SIMON, H. D., AND W. WANG, L. Thick-restart Lanczos method for electronic structure calculations. *Journal of Computational Physics* 154, 1 (Sept 1999), 156–173.
- [155] WU, K., AND SIMON, H. Thick-restart lanczos method for symmetric eigenvalue problems. In *Solving Irregularly Structured Problems in Parallel*, A. Ferreira, J. Rolim, H. Simon, and S.-H. Teng, Eds., vol. 1457 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998, pp. 43–55. 10.1007/BFb0018526.
- [156] XILINX. *Virtex-5 FPGA User Guide*, 2010.
- [157] XILINX CORP. Core generator overview, Sept 2010. http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/isehelp_start.htm#cgn_c_overiew.htm.
- [158] ZHANG, L., ZHANG, Y., AND ZHOU, W. Tradeoff between approximation accuracy and complexity for range analysis using affine arithmetic. *J. Signal Process. Syst.* 61 (December 2010), 279–291.
- [159] ZHAO, Z., AND LEESER, M. Precision modeling and bit-width optimization of floating-point applications. In *High Performance Embedded Computing* (2003), pp. 141–142.
- [160] ZHUO, L., MORRIS, G. R., AND PRASANNA, V. K. Designing scalable FPGA-based reduction circuits using pipelined floating-point cores. In *Proc. Int. Symp. on Parallel and Distributed*

Bibliography

- Processing Symposium - Workshop 3* (Washington, DC, USA, 2005), IEEE Computer Society, p. 147.1.
- [161] ZHUO, L., MORRIS, G. R., AND PRASANNA, V. K. High-performance reduction circuits using deeply pipelined operators on FPGAs. *IEEE Trans. Parallel Distrib. Syst.* 18, 10 (2007), 1377–1392.
- [162] ZHUO, L., AND PRASANNA, V. K. Sparse matrix-vector multiplication on FPGAs. In *Proc. Int. Symp. on Field-programmable gate arrays* (New York, NY, USA, 2005), ACM, pp. 63–74.