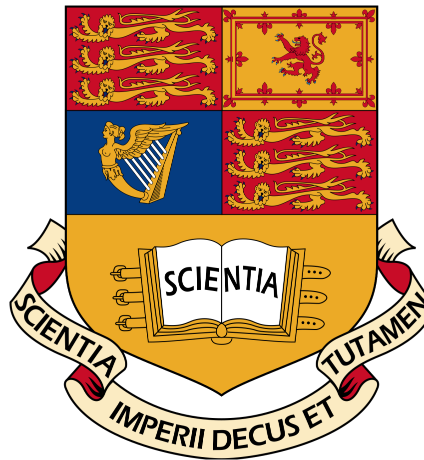
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2017



Project Title:	**Fused Arithmetic Analysis for Efficient Hardware Datapath**
Student:	**Eusebius M. Ngemera**
Email:	**me@eusebius.tech**
Course:	**MEng Electrical & Electronic Engineering**
Project Supervisor:	**Professor George A. Constantinides**
Second Marker:	**Professor Peter Y. K. Cheung**

Abstract

During high-level synthesis (HLS) more often than not, standard discrete-arithmetic units are used to synthesise C-like arithmetic programs onto a field-programmable gate array (FPGA). In this project, I expand on the open-source HLS tool 'Structural Optimisation of Arithmetic Programs' (SOAP) to consider fused arithmetic units.

SOAP rewrites a program to produce a set of equivalent programs that are Pareto-optimal in resource usage (area), numerical accuracy and latency. In this project, just the area and numerical accuracy of individual arithmetic expressions are evaluated. The fused units implemented are the 3-input adder, fused multiply-add (FMA) and constant multiplier, all acting on floating-point numbers. SOAP's power of expression transformation is harnessed to expose non-obvious uses of these fused units.

Improvements achieved in some of the inner expressions of PolyBench and Livermore Loops, for single-precision floating-point, are up to 1.13× in area without degrading accuracy, and up to 1.4× in numerical accuracy at an area cost of up to 2.6×. It is shown that the accuracy improvement can be arbitrarily large under certain conditions.

Acknowledgements

Standing on the shoulders of giants. I would like to thank my exceptional project supervisor, Professor George A. Constantinides, for his support, guidance and insights. Xitong Gao and his work on *SOAP*, have been vital to this project's accomplishments. Support from Dr Thomas J. W. Clarke is also very much appreciated.

I would also like to thank my family and friends for their unending encouragement. Last but not certainly not least, my mother for her many forms of support showing unconditional love.

God's grace is good.

Table of Contents

	Page
List of Tables	**vii**
List of Figures	**ix**
List of Source Codes	**xi**
1 Introduction	**1**
1.1 High-Level Synthesis	1
1.2 Introduction to SOAP	2
1.3 Objective	2
2 Background	**5**
2.1 Floating-Point Arithmetic	5
2.1.1 Accuracy Issues	6
2.2 Structural Optimisation of Arithmetic Programs (SOAP)	6
2.2.1 Estimating Area	6
2.2.2 Estimating Error	8
2.2.3 Expression Transformation	10
2.2.4 An Example	10
2.3 Floating-Point Fused Arithmetic	11
2.3.1 Three-Input Adder	12
2.3.2 Fused Multiply-Add	13
2.3.3 Algorithmic Alternatives	14
2.4 Requirements	15
3 Analysis and Design	**17**
3.1 Three-Input Adder	17
3.1.1 Area	17
3.1.2 Error	17
3.1.3 Transformations	18
3.2 Fused Multiply-Add	19

3.2.1 Area 20
3.2.2 Error 21
3.2.3 Transformations 21
3.3 Constant Multiplier 21
3.3.1 Area 22
3.3.2 Error 22
3.3.3 Transformations 22

4 Implementation **23**
4.1 Adding Operators 24

5 Testing **25**
5.1 Three-Input Adder 25
5.2 Fused Multiply-Add 28
5.2.1 The Single-Use FMA 29
5.3 Constant Multiplier 30

6 Results **33**
6.1 Single-Precision, Multiple-Use FMA 33
6.2 Single- and Double-Precision, Single- and Multiple-Use FMA 37

7 Evaluation **39**
7.1 Against Varying Precision 39
7.2 Against SOAP Version 3 40

8 Conclusions **43**
8.1 Further Work 43

A User Guide **45**
A.1 Installing 45
A.2 Usage 46

B Benchmarks **47**
B.1 PolyBench 47
B.2 Livermore Loops 49

Bibliography **51**

LIST OF TABLES

TABLE **Page**

6.1 Summary of benchmark results for single precision and with the multiple-use implementation of the FMA 35
6.2 Aggregate of benchmark results for single and double precision, and for the two types of FMA units 38

List of Figures

Figure **Page**

2.1 Expressions equivalent to $0.2 \times (a+b+c+d+e)$ optimised for single precision ($w_F = 23$) 11
2.2 Patent schematics for a 3-input floating-point adder [1] . . . 12

3.1 A comparison of the LUT usage of a 3-input addition between implementing with two 2-input adders and one 3-input adder . . . 18
3.2 A comparison of the absolute error of a 3-input addition between implementing with two 2-input adders and one 3-input adder . . . 19
3.3 Parameters for FloPoCo's fixed-point accumulator, `LongAcc` [2] . . . 20

5.1 Frontiers before and after including the 3-input adder for 3-operand addition at single precision . . . 26
5.2 Frontiers before and after including multiple-use FMA for a simple expression at single precision . . . 28
5.3 Frontiers before and after including single-use FMA for a simple expression at single precision . . . 30
5.4 Frontiers before and after including the constant multiplier, for a π multiplier at single precision input/output . . . 31

6.1 Frontiers for `fdtd-2d_1` at single precision, and with the multiple-use FMA . . . 36
6.2 Frontiers for `jacobi-1d` at single precision, and with the multiple-use FMA . . . 36
6.3 Frontiers for `2mm_2` at single precision, and with the multiple-use FMA . . . 37
6.4 Full closure . . . 37
6.5 Greedy trace . . . 37
6.6 Frontiers for `state_frag` at a transformation depth of 3, at single precision and with the multiple-use FMA . . . 37

7.1 Frontiers for varying the precision of the frontier of `seidel` at single precision, and with the multiple-use FMA . . . 40
7.2 Frontiers for `seidel` at single precision, and with the multiple-use FMA, compared to a matched and modified version 3 of SOAP . . . 41

List of Source Codes

Listing		Page
1	Kahan Compensated Summation Algorithm pseudo code	15
2	Snippet from the `2mm` kernel in PolyBench [3]	34

Chapter 1

Introduction

1.1 High-Level Synthesis

In the field of high-performance computing, the graphics processing unit (GPU) has long been the gold standard for accelerating programs with large amounts of data [4]. However, advancements in Silicon technology are bringing field-programmable gate arrays (FPGAs) more to the scene. FPGAs bring versatility in the form of reconfigurable digital hardware allowing them to be used as accelerators of floating-point arithmetic, customised to the desired application.

However, FPGAs have been traditionally configured using cumbersome hardware description languages like VHDL[1] and Verilog on a gate level or a register-transfer level (RTL) whilst GPUs have tended to be significantly easier to program. High-level synthesis (HLS) brings an algorithmic level to hardware design. Manohar *et al.* [5] describe HLS tools as handling the micro-architecture and transforming "untimed or partially timed functional code into fully timed RTL implementations, automatically creating cycle-by-cycle detail for hardware implementation."

HLS has given hardware designers the abstraction needed to easily create hardware from C-like code and focus on algorithmic optimisations rather than low-level implementation details. HLS has also brought software developers to hardware design due to lower learning curves when coming from software programming. However, designers are "limited to the set of numerical primitives provided by HLS vendors" as noted by Thomas in [6] while investigating a method for automatically creating high-performance floating-point function approximations.

FPGAs are effectively reconfigurable alternatives to application-specific integrated circuits (ASICs) which are fixed—and hence seldom used as application accelerators—but provide the best power efficiency. However, their development succumbs to longer periods and with higher costs.

[1] Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

1.2 Introduction to SOAP

'Structural Optimisation of Arithmetic Programs' (SOAP) is an open-source, Python-coded tool by Gao *et al.* [7–9] that optimises three key constraints for a hardware designer as part of a high-level synthesis flow. SOAP automatically restructures numerical C programs to optimise for latency, numerical accuracy and resource usage when synthesised onto an FPGA. Current HLS tools cannot safely make these rewrites because they do not consider the numerical correctness and performance implications of such restructuring.

An example of restructuring that SOAP is more capable of is loop pipelining since it considers inter-iteration dependencies amongst variables. The design space of equivalent structures of a program is efficiently analysed to produce a three-dimensional frontier of Pareto-optimal, candidate restructured programs. This provides a hardware designer with the freedom to trade off the three properties—latency, accuracy and resource usage—depending on the hardware's application and design constraints.

1.3 Objective

With version 2 of SOAP, the current one, the arithmetic units that can be synthesised are the 2-input adder/subtractor and 2-input multiplier of either fixed-point or floating-point numbers, as well as more advanced floating-point operations, namely division and exponentiation.

This project aims to discover and analyse improvements that can be made by fused arithmetic units during high-level synthesis through expanding the capabilities of SOAP. Namely, the fused arithmetic units in this project are the floating-point 3-input adder, fused multiply-add (FMA) and constant multiplier. This project limits its scope to exclude latency, and only analyse area in the form of the number of lookup tables (LUTs) needed and numerical accuracy in the form of maximum absolute error. There is also a focus on individual, nested expressions as opposed to entire programs.

Quantitatively, the objective is to improve the area and absolute error of the set of Pareto-optimal equivalent expressions that SOAP discovers. This set should contain one or more candidates that have been transformed with at least one fused arithmetic unit.

In the process, this project aims to answer some questions: What is the extent of changes to the frontier of the design space, if any, with these fused units? Are any increases in duration of the expanded search for equivalent expressions worthwhile? How does the approach of fusing arithmetic units compare to that of varying precision in order to achieve better area or accuracy?

Structure of the Report

Chapter 2 gives a background to floating-point arithmetic, SOAP and fused arithmetic units. Chapter 3 covers the modelling of the fused units. Chapter 4 provides details of the implementation. Chapter 5 covers verification of the functional correctness of each fused arithmetic unit's implementation. Chapter 6 gathers and explains results from benchmark expressions.

Chapter 7 compares the changes due to the fused arithmetic units to current design-space search methods of improving area and numerical accuracy. Chapter 8 concludes and discusses further work. A user guide to installing and using the implementation is included in Appendix A.

Chapter 2

Background

2.1 Floating-Point Arithmetic

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) [10] defines computation for floating-point numbers. The standard describes not only the data format but also surrounding aspects such as rounding rules and exception handling. A binary (base 2) floating-point number x is represented as in equation 2.1. S is a single bit that determines the sign of the number to be either positive or negative. The mantissa, or significand, is the number $1.F$, which lies in the range $[1,2)$ and contains the fractional binary digits F. The exponent E is represented as a non-negative integer from which a bias is subtracted.

$$x = (-1)^S \times 1.F \times 2^{E-bias} \tag{2.1}$$

The sign bit S, exponent E and fraction F are packed together in that order. Hereafter, w_E is defined as the width in number of bits of the exponent E and w_F is defined as that of F. Also, $q := E - bias$ is the true exponent of x.

The value of the exponent bias is dependent on the exponent width: $bias := 2^{w_E-1} - 1$. The true exponent q lies in the range of integers $[-(2^{w_E-1}-2), 2^{w_E-1}-1] \equiv [-(bias-1), bias]$. The range of q excludes the boundaries of E, i.e. $E = 0$ and $E = 2^{w_E} - 1$. These boundary exponents are used to represent ± 0, $\pm\infty$, NaN's (not a number) and subnormal numbers. Subnormal number representation allows numbers in the underflow interval around zero: $(-2^{-(2^{w_E}-2)}, 2^{-(2^{w_E}-2)})$. Subnormal numbers are not considered in SOAP nor this project as explained in section 2.2.1.1.

IEEE defines the standard binary precisions *half*, *single*, *double*, *quadruple* and *octuple* ranging from 16 to 256 bits in total width. The most common floating-point formats are the 32-bit single precision ($w_E = 8$, $w_F = 23$) and the 64-bit double precision ($w_E = 11$, $w_F = 52$) binary representations.

Floating-point trades off range for precision when compared to standard fixed-point representation. Floating-point can give a higher range that is dynamic but at the cost of exponentially increasing round-off error due to the difference between consecutive numbers amplifying with the scale.

2.1.1 Accuracy Issues

A good example of an accuracy issue is when summing a sequence of non-negative floating-point numbers. The summation in equation 2.2 is from Higham [11]. The sequence of summed numbers is in a polynomially decreasing order and Higham notes that reversing the order so that the sequence is increasing makes the error 650 times smaller.

$$\sum_{n=1}^{10,000} n^{-2} \tag{2.2}$$

When sequences contain both negative and positive numbers, generally there are still accuracy issues even with summing in order of magnitude. Higham suggests that the negative numbers should be summed separately to the positives and then the two sums combined: $S_n = S_- + S_+$. This has been an introduction to floating-point accuracy issues and the matter continues to be discussed throughout the rest of the report.

2.2 Structural Optimisation of Arithmetic Programs (SOAP)

Gao *et al.* [7, p. 2] note that SOAP has been the first tool to "perform trade-off optimisation between numerical accuracy and resource usage by varying the *structure* of arithmetic programs." SOAP relies on knowledge of the input variable's upper and lower bounds to guide restructuring towards optimality.

This project aims to set the foundation for incorporating fused arithmetic units into the current version of SOAP by first building on version 1 of SOAP. Versions 2 and 3 added the ability to analyse full C programs with loops, instead of individual expressions, as well as analysing and optimising for latency. Version 1 of SOAP is still valid and relevant to this project as it has the core ability of rewriting arithmetic expressions to optimise for area and accuracy.

The input to version 1 of SOAP is an arithmetic expression and the range of values for its variables, i.e. the minimum and the maximum of possible values. The output is a set of expressions that are equivalent in real arithmetic and that are Pareto-optimal in area (number of LUTs) and maximum absolute error.

This section covers how the area and error are estimated, how the expression is transformed and an example is given.

2.2.1 Estimating Area

The target platform in version 1 of SOAP is Xilinx's *Virtex6 XC6VLX760* FPGA. Xilinx [12] describes their FPGA device family as offering "the best solution for addressing the needs of high-performance

logic designers". The basic element of an FPGA is the configurable logic block (CLB) with slices which usually contain lookup tables (LUTs) and flip-flops. In SOAP, area is measured in terms of the number of LUTs since they are the core component to implementing Boolean functions and digital logic. An n-input LUT can implement a Boolean equation of n variables.

To calculate the number of LUTs required, sub-expressions are given unique labels but maintaining that the same operation in different parts of the expression with the same parameters and inputs get the same label. From this collection of labels, unique operators required are counted and multiplied by the number of LUTs required for the matching operator.

Because value intervals are known throughout the datapath, the exponent size of a component can be set to one such that no overflow occurs (at neither its inputs nor its output), as opposed to that set by the final output value range. For example, if an addition's inputs nor output do not need any more than 5 bits for the exponent size then an adder with an exponent size $w_E = 5$ is synthesised. This need for operators to be specific to a particular exponent and mantissa size requires a capable floating-point core generator, and this is where FloPoCo comes in.

2.2.1.1 FloPoCo

FloPoCo is a software tool by Dinechin *et al.* [13] that generates arithmetic cores. Its developers note that "all FloPoCo operators are fully parametrised in precision, so that an application may use just the precision it needs, and accurate to the last bit, so that wires don't carry meaningless noise." FloPoCo provides a VHDL hardware description of a customised floating-point arithmetic unit, for example it can generate a floating-point multiplier of single precision ($w_F = 23$) with an exponent size w_E of 5 instead of the standard 8 bits. FloPoCo version 2.5.0 is used in this project.

FloPoCo's developers also demonstrated their generator "ranking higher than the results obtained by FPLibrary 2 and not far from the operators generated with Xilinx CoreGen 3"[2]. Crucially, they note that the generate VHDL code is portable and target independent, while Xilinx's CoreGen only works with their own line of FPGAs. However, FloPoCo does not support subnormal floating-point numbers because they are viewed as a rare edge case.

Given the aforementioned VHDL file from FloPoCo, Xilinx's ISE Design Suite (version 14.7) is used to synthesise the VHDL file to produce a device utilisation summary—specific to the Virtex-6 target platform—which includes the number of LUTs needed before place-and-route. Place-and-route is the process of mapping the circuit description to the FPGA's physical resources. In tests, Gao *et al.* in [7, p. 8] note that SOAP's area estimation is a 6.1% over-approximation and the worst case over-approximation is 7.7% of the actual number of LUTs.

The SOAP installation includes pre-synthesised and precomputed values for the number of LUTs for the floating-point adder and multiplier of varying w_E and w_F, which means FloPoCo and Xilinx's ISE Design Suite are not needed during runtime.

2.2.1.2 Exponent Width Required

Given a value interval $[x_1, x_2]$, the range of exponents $[^{\min}q, ^{\max}q]$ needs to be determined. Then given this range. the minimum exponent size needed to hold the interval of exponents is calculated. Let x be one of the (non-zero) interval bounds. The exponent of x is $q_x = \lfloor \log_2(|x|) \rfloor$. The minimum exponent size w_E required to include a true exponent q is given in equation 2.3. These arise from knowing the range of valid true exponents for a given exponent width: $q \in [-(2^{w_E-1} - 2), 2^{w_E-1} - 1]$.

$$w_E = \begin{cases} \lceil 1 + \log_2(2 - q) \rceil, & \text{if } q \leq 0 \\ \lceil 1 + \log_2(1 + q) \rceil, & \text{otherwise} \end{cases} \tag{2.3}$$

It was discovered that in the implementation of version 1 of SOAP, only the upper bound of the exponent was evaluated and used to determine the minimum exponent size. SOAP was updated to also check the lower bound and hence use the maximum of the two computed exponent sizes.

It was also observed that when a value interval included zero then SOAP would give the minimum synthesisable exponent size of 5. Generally, this is not correct as a non-singular interval including zero means the lower exponent bound is effectively $^{\min}q$ (due to infinitesimal non-zero numbers to be represented), which is circularly dependent on w_E. IEEE defines that the standard exponent size w_E for a non-standard w_F needs to match that of the next-up standard w_F [10]. This is illustrated in equation 2.4.

$$w_E = \begin{cases} ^{\text{half}}w_E = 5, & \text{if } \quad 0 < w_F \leq \ ^{\text{half}}w_F = 10 \\ ^{\text{single}}w_E = 8, & \text{if } \ 10 < w_F \leq \ ^{\text{single}}w_F = 23 \\ ^{\text{double}}w_E = 11, & \text{if } \ 23 < w_F \leq \ ^{\text{double}}w_F = 52 \\ ^{\text{quadruple}}w_E = 15, & \text{if } \ 52 < w_F \leq \ ^{\text{quadruple}}w_F = 112 \\ ^{\text{octuple}}w_E = 19, & \text{if } 112 < w_F \leq \ ^{\text{octuple}}w_F = 236 \end{cases} \tag{2.4}$$

2.2.2 Estimating Error

Numerical accuracy is determined by estimating the maximum absolute error. An interval representing possible floating-point values of an input variable propagates through the expression's datapath. This results in a floating-point interval for the output values of the expression. From bottom to top, the ranges of absolute error of sub-expressions are evaluated. An absolute error range is in the form of an interval of real (rational) numbers. The absolute error applies to the interval bound with the largest magnitude.

In this project, $\epsilon : \textbf{Expression} \to \mathbb{R}$ is defined to be a function that gives the maximum magnitude of the absolute error of an expression. The function $\delta : \textbf{Interval}_{\mathbb{F}} \to \mathbb{R}$ is defined to give the maximum magnitude of the round-off error for an interval of floating-point numbers, where the set $\mathbb{F}$ is the set of possible floating-point numbers for a particular w_E and w_F. These definitions are slight

simplifications of the implementation as both ϵ and δ output $\mathbf{Interval}_{\mathbb{R}}$, an interval that usually spans between the negative and positive of the same absolute value.

$\delta([x_1, x_2])$, the maximum round-off error of floating-point interval $[x_1, x_2] \in \mathbf{Interval}_{\mathbb{F}}$ is as given in equation 2.6. If evaluating the round-off error of a constant in the expression then equation 2.5 is used instead, where $fl(x)$ provides the floating-representation of x after rounding. Unit of the last place (ulp) is the weight of the least significant bit in the mantissa. Muller [14] notes that ulp can have different interpretations but this project continues the definition that SOAP uses in equation 2.7.

$$\delta(x) = x - fl(x), \quad x \in \mathbb{R} \tag{2.5}$$

$$\delta([x_1, x_2]) = \frac{1}{2} \times ulp(\max(|x_1|, |x_2|)) \tag{2.6}$$

$$ulp(x) = 2^{-w_F} \times 2^{(E_x - bias)} = 2^{q_x - w_F} \tag{2.7}$$

The 2-input addition $(a+b)$ is taken as an example and assuming its variables $a, b \in \mathbb{R}$ have an associated error from their real values, Δ_a and Δ_b respectively. Evaluations as shown in equation 2.8 reveal the error of the operation is the sum of the input errors added with the error from rounding-off the result. Given x to be a real number, $fl(x)$ is its floating-point representation after rounding: $fl : \mathbb{R} \to \mathbb{F}$.

$$\begin{aligned} &fl(\,(a+\Delta_a) + (b+\Delta_b)\,) \\ =&(a+b) + (\Delta_a + \Delta_b + \delta(\,[a+b+\Delta_a+\Delta_b]\,)) \end{aligned} \tag{2.8}$$

The maximum absolute error can be expressed as in equation 2.9. The floating-point interval $[a+b+\Delta_a+\Delta_b]$ from equation 2.8 is written simply as $[a+b]$. The implementation in SOAP performs the floating-point interval addition $[a]+[b] = [a+b] \in \mathbb{F}$ by emulating a particular precision and therefore including Δ_a and Δ_b as explained in chapter 4. $\delta([a+b])$ represents the round-off error of the floating-point interval arising from expression $(a+b)$.

$$\begin{aligned} &\epsilon(a+b) \\ =&\epsilon(a) + \epsilon(b) + \delta([a+b]) \end{aligned} \tag{2.9}$$

Evaluating the error of 2-input multiplication in equation 2.10 reveals its maximum absolute error can be expressed as in equation 2.11.

$$\begin{aligned} &fl(\,(a+\Delta_a) \times (b+\Delta_b)\,) \\ =&(a \times b) + (\Delta_a \times \Delta_b + \Delta_a \times b + \Delta_b \times a + \delta(\,[\,(a+\Delta_a) \times (b+\Delta_b)\,]\,)) \end{aligned} \tag{2.10}$$

$$\begin{aligned} &\epsilon(a+b) \\ =&\epsilon(a) \cdot \epsilon(b) + \epsilon(a) \cdot \max(|b|) + \epsilon(b) \cdot \max(|a|) + \delta([a \times b]) \end{aligned} \tag{2.11}$$

2.2.3 Expression Transformation

Rules of real arithmetic are used to restructure expressions. These are associativity, distributivity and commutativity as illustrated in equation sets 2.12, 2.13 and 2.14 respectively.

$$\begin{aligned}(a+b)+c &\Longleftrightarrow a+(b+c)\\ (a\times b)\times c &\Longleftrightarrow a\times(b\times c)\end{aligned} \tag{2.12}$$

$$\begin{aligned}a+b &\Longleftrightarrow b+a\\ a\times b &\Longleftrightarrow b\times a\end{aligned} \tag{2.13}$$

$$a\times(b+c) \Longleftrightarrow (a\times b)+(a\times c) \tag{2.14}$$

Furthermore, reduction rules in equation set 2.15 help to maintain a unique set of expressions. An expression with inputs that are all constants is evaluated to a value, for example $(2*3)\Rightarrow 6$.

$$\begin{aligned}a\times 1 &\Rightarrow a\\ a+0 &\Rightarrow a\end{aligned} \tag{2.15}$$

SOAP recursively gathers a set of transformed expressions until the depth limit is reached or there are no new additions to the set. Expressions can be too complex to perform a full transitive closure as memory runs out. Gao *et al.* [7, p. 7] conclude with a greedy trace to efficiently generate equivalent expressions on the Pareto frontier. However, in this project, the aim is to evaluate full closures unless otherwise stated.

2.2.4 An Example

The expression and interval ranges in equation 2.16 from the `seidel` benchmark kernel (introduced in chapter 6) are analysed with SOAP.

$$\begin{gathered}0.2\times(a+b+c+d+e)\\ a\in[0,1], b\in[0,1], c\in[0,1], d\in[0,1], e\in[0,1]\end{gathered} \tag{2.16}$$

42 equivalent expressions are discovered and all of these occupy just four points in the plot in figure 2.1, i.e. there are only four unique area–error points in the design space. The restructured expression $((((d+e)+b)+(a+c))\times 0.2)$ is discovered to achieve lower absolute error whilst using the same number of LUTs. This is solely due to balancing the additions such that error at any of the inputs propagates through at most 3 adders instead of 4 originally. The expression $((((a+c)+b)\times 0.2)+((d+e)\times 0.2))$ which further balances the operators, has the lowest absolute error but at an area cost due to the additional multiplier now needed. Expression $(((((a+b)+c)+d)\times 0.2)+(0.2\times e)$ is in the mid-right of the plot and is non-optimal because it adds a multiplier and ends up only partially balancing the expression.

The term *Pareto frontier* refers to the set of expressions that are Pareto-optimal. In multi-objective optimisation—as here with area and error—there can be conflicting points where no one point can

be said to be worse than any other. For example, the expression in the bottom right improves the absolute error of that in the bottom left but with a worse area. However, the expression in the top left is definitively worse than that in the bottom left because it maintains the same area but degrades the error.

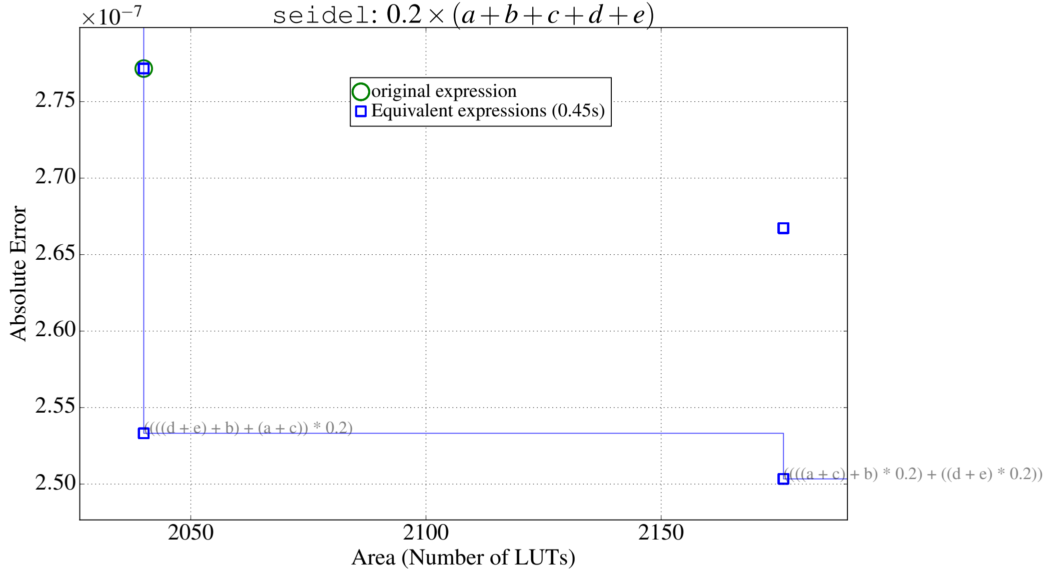


Figure 2.1: Expressions equivalent to $0.2 \times (a+b+c+d+e)$ optimised for single precision ($w_F = 23$)

2.3 Floating-Point Fused Arithmetic

Fused arithmetic units aim to provide improvements in latency, accuracy and resource usage by eliminating redundant circuitry present in standard, discrete arithmetic, floating-point core based implementations. Most of this redundancy can arise from normalisation and rounding stages at the end of floating-point arithmetic which adds area, increases latency and reduces accuracy when inside a datapath [15].

Swartzlander *et al.* implemented a single-precision radix-2 Fast Fourier Transform (FFT) butterfly using two fused arithmetic units: a floating-point fused multiply-add and a floating-point fused add-subtract [16] [17]. The result was an improvement in latency of 15% (1.18×) and a reduction of 39% (1.64×) in area (in a 45nm process) compared to implementing using conventional two-input floating-point adders and multipliers. The numerical result of the fused butterfly unit was also more accurate because of fewer intermediate rounding operations. Improvements of up to 3.3× in latency and 19.7× in area (90nm technology) are made on an application-specific integrated circuit (ASIC) by Smith *et al.* [15] and Langhammer [18].

Langhammer [18] proposes fusing the entire datapath by optimising at the "expression and not

operator level; this is achieved by reducing each operator into denormalization, operation, and normalization stages, and then only using the minimal combination of stages at any node. Exception handling is also done in parallel with the datapath, rather than being applied at each node." He notes that this results in a typical 50% logic, latency, and power reduction.

2.3.1 Three-Input Adder

One of the basic fused arithmetic units is a floating-point adder with three operands and performs an addition/subtraction without intermediate result truncation: $fl(a \pm b \pm c)$. Given x to be a real number, $fl(x)$ is its floating-point representation after rounding. This results in rounding errors getting approximately halved [11], compared to executing $fl(fl(a \pm b) \pm c)$, i.e. two distinct operations.

Fukumura *et al.* hold a patent with the schematic in figure 2.2 [1]. The pre-processing circuit aligns the 3 significands as well as determining the maximum exponent difference. The normalization circuit ensures the final result has one leading 1. During addition, there may be two leading 1's and during subtraction there may be leading 0's in the significand of the result. Addition implemented with one 3-input adder normalises and truncates the result once thus reducing the error and achieving a shorter critical path than with two discrete 2-input adders.

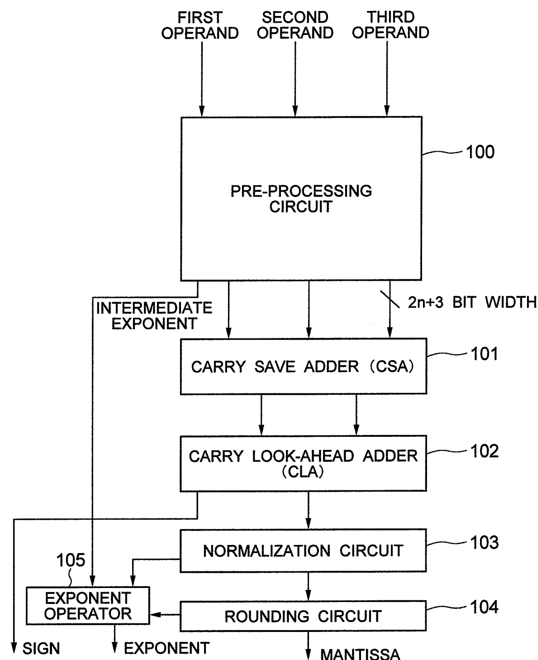


Figure 2.2: Patent schematics for a 3-input floating-point adder [1]

The expression in equation 2.17, which always equals -2^{-k} in real arithmetic, will now be analysed. The value of k is set such that the difference between exponents is large enough so that the

last bit of $(1+2^{-k})$ cannot be stored in the mantissa, i.e. $k > w_F$ (assuming no use of subnormal floating-point numbers).

$$2^0 - 2^{-k} - 2^0 = -2^{-k} \tag{2.17}$$

There are two possible orders of computation using 2-input adders as illustrated in equations 2.18 and 2.19.

$$fl\Big(fl\Big(2^0 - 2^{-k}\Big) - 2^0 \Big) = fl\big(2^0 - 2^0 \big) = fl(0) \tag{2.18}$$

$$fl\Big(2^0 + fl\Big(-2^{-k} - 2^0\Big) \Big) = fl\big(2^0 - 2^0 \big) = fl(0) \tag{2.19}$$

Given this order of addition without commutativity, none of the possible uses of the 2-input adder give a correct answer. The implication of this can be when comparing around zero. Using 2-input adders results in making an equal-to-zero comparison whereas using the 3-input adder would have the correct comparison result of less-than-zero. This can cause a finite-state machine to deviate from the expected path, and perhaps even result in an infinite loop in some applications. Numerical instability to a control system would render it unstable. When the expression in equation 2.17 is computed using a 3-input adder then there is no error due to intermediate rounding, and similarly for any multiple-operand floating-point adder with more than 2 inputs.

$$fl\Big(2^0 - 2^{-k} - 2^0 \Big) = fl\Big(-2^{-k}\Big) \tag{2.20}$$

2.3.2 Fused Multiply-Add

Another example of a fused arithmetic unit is the fused multiply-add (FMA), also known as a multiply-accumulate (MAC). An FMA performs multiplication followed by addition/subtraction without intermediate result truncation: $(a \times b) \pm c$. Rounding errors can get approximately halved [11]. The inner product between two vectors $\mathbf{x}$ and $\mathbf{y}$, both of length n, can be expressed as in equation 2.21.

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n \tag{2.21}$$

This result can be calculated with n rounding errors instead of $2n-1$ arising from the two operations (multiplication and addition) at each iteration pass. Any applications dominated by a form of an inner product will reap benefits from using FMAs. Inner products dominate matrix multiplications. Some examples of applications that could see significant program transformations and hence benefit from FMA units include optimisation, digital filtering as well as artificial neural networks.

Additionally, FMAs can be used to compute an exact representation of the product between two floating-point numbers. Higham *et al.* demonstrates that given two floating-point numbers, x and y,

the exact representation of their product, $x \cdot y$, can be computed as shown in equation set 2.22 [11].

$$\begin{aligned} \hat{a} &= fl(x \cdot y) \\ \hat{b} &= fl(x \cdot y - \hat{a}) \\ x \cdot y &= \hat{a} + \hat{b} \end{aligned} \tag{2.22}$$

$\hat{b}$ has to be computed with only one rounding-off error and hence can only be done by a fused arithmetic unit performing a multiplication followed by addition/subtraction.

2.3.2.1 Pitfalls

If an expression originally is $(a \times b) + (c \times d)$ then there are two equivalence possibilities using an FMA unit:

$$\begin{gathered} fma(a,\ b,\ fl(c \times d)) \\ fma(c,\ d,\ fl(a \times b)) \end{gathered} \tag{2.23}$$

$fma(x, y, z)$ is a fused multiply-add unit performing $fl(x \times y + z)$. Which of the two multiplications should be computed and rounded first? This is where FMAs may come with a potential danger[11, p. 47].

In solving the quadratic $f(x) = ax^2 + bx + c$ for x, the square root of the discriminant, $\sqrt{b^2 - 4ac}$, needs to be computed. With correctly rounded arithmetic, monotonicity in rounding makes sure $fl(b^2) - fl(4ac) \geq 0$ and hence the computed discriminant will always be a non-negative number for all non-negative $b^2 - 4ac$ in real arithmetic. On an FMA however, $fl(fl(b^2) - 4ac)$ would be computed instead and might break the monotonicity relation. This could happen, for example, if $b^2 = 4ac$ in real arithmetic and $fl(b^2) < b^2$ resulting in $fl(b^2) < 4ac$.

As Kahan [10, p. 5] puts it, “therefore Fused MACs cannot be used indiscriminately; there are a few programs that contain a few assignment statements from which Fused MACs must be banned." This is where SOAP has the potential, at its core, able to analyse equivalent expressions for improvements in latency, accuracy and resource usage. When analysing the aforementioned quadratic equation discriminant, SOAP should be able to determine errors such as invalid square rooting of a negative number in the quadratic equation discriminant above as well as choosing the best way to perform FMA in evaluating $(a \times b) + (c \times d)$ to reduce rounding error.

2.3.3 Algorithmic Alternatives

Apart from fused arithmetic units and increasing precision, numerical accuracy can be improved in other ways. For example, Kahan’s compensated summation algorithm [19] reduces round-off error when summing a sequence of floating-point numbers by using a compensation at each iteration. The algorithm keeps a running compensation of low-order bits. Pseudo code for the algorithm is included in listing 1 and comes from [20, p. 791].

```
sum = 0
compensation = 0

for i = 1:n
    temp = sum
    y = x[i] + compensation
    sum = temp + y
    compensation = (temp - sum) + y
end
```

Listing 1: Kahan Compensated Summation Algorithm pseudo code

The algorithm reduces the worst-case error from $O(n)$ to $O(1)$, i.e. now independent of the input size. What should also be noted is that there are now 4 additions/subtractions at each iteration instead of just 1, resulting in more resource usage. This algorithm could be avoided entirely by using higher precision arithmetic units during computation.

2.4 Requirements

Continuing on, the main project aims established are:

- Incorporate at least one fused arithmetic unit into SOAP's models
- Verify presence of fused arithmetic units in equivalent expressions
- Report on changes to the Pareto frontier for various benchmark programs
- State conditions that make the fused unit(s) advantageous

CHAPTER 3

Analysis and Design

3.1 Three-Input Adder

Out of the FloPoCo units available to synthesise, the `FPAdder3Input` unit is the only one with the the functionality needed.

3.1.1 Area

Figure 3.1 shows a comparison of the number of LUTs needed to implement 3-operand addition using either two 2-input adders or one 3-input adder, for varying mantissa size as well as exponent size. It is evident that the 3-input adder does not save LUT usage. This can be attributed to the barrel shifter, a digital circuit to shift data by a number of bits using only combinatorial logic. As previously seen in the schematics in figure 2.2, a pre-processing circuit acts on the 3 inputs to align them and this needs a barrel shifter. However, barrel shifters generally have circuitry complexity that increases exponentially with the size of the input, as evident by the exponential increase when varying the mantissa size. On the other hand, the exponent size has little to no effect in the LUT usage of both adders.

3.1.2 Error

Continuing on from the principles set in section 2.2.2, the maximum absolute error for a 3-input adder is modelled as in equation 3.1. $add3(a,b,c)$ is equivalent to $(a+b+c)$ in real arithmetic. It should be noted that this model is the same as that using two 2-input addition but without any

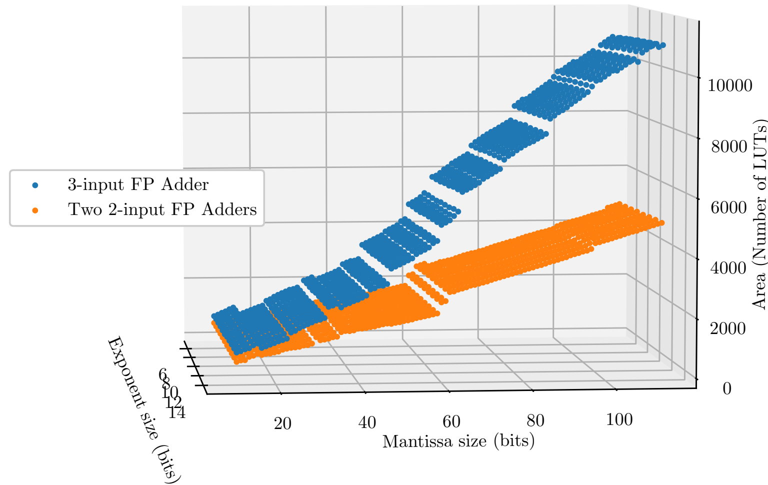


Figure 3.1: A comparison of the LUT usage of a 3-input addition between implementing with two 2-input adders and one 3-input adder

intermediate round-off error, $\delta([a+b])$ for example.

$$\begin{aligned} &\epsilon(\ add3(a,b,c)\ ) \\ =&\epsilon(a)+\epsilon(b)+\epsilon(c)+\delta([a+b+c]) \end{aligned} \tag{3.1}$$

From the error model, figure 3.2 is obtained which illustrates the error improvement for varying mantissa size. There's exponential decay of absolute error with increasing mantissa size, for both adders, so the difference between them is also exponentially decaying. This difference is due to the lack of intermediate round-off error with a 3-input adder. The exponential decay is as expected because round-off error decays exponentially with w_F.

In a standard summation of n numbers using the 2-input adder, there will be $n-1$ operations and hence $n-1$ round-off errors. With a 3-input adder, the number of operations is $\lfloor \frac{n}{2} \rfloor$ and therefore effectively halves the number of round-off errors.

3.1.3 Transformations

To allow 3-operand expressions to be restructured to use the 3-input adder, transformation rules need to be added to SOAP. These are illustrated in equation set 3.2. Equation 3.3 shows the commutativity rule.

$$\begin{aligned} (a+b)+c &\Longleftrightarrow add3(a,b,c) \\ a+(b+c) &\Longleftrightarrow add3(a,b,c) \end{aligned} \tag{3.2}$$

$$add3(a,b,c) \Leftrightarrow add3(a,c,b) \Leftrightarrow add3(b,a,c) \Leftrightarrow add3(b,c,a) \Leftrightarrow add3(c,a,b) \Leftrightarrow add3(c,b,a) \tag{3.3}$$

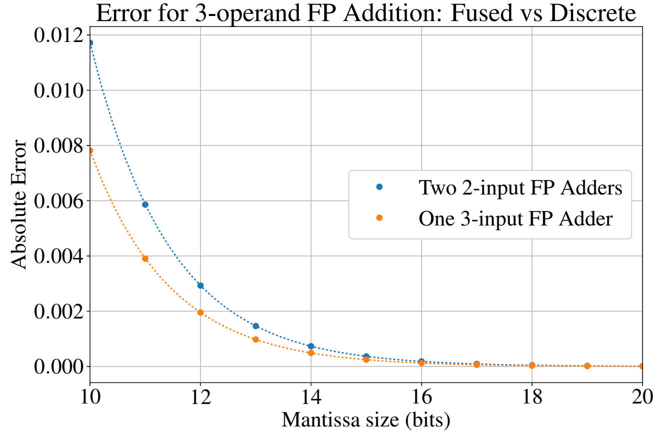


Figure 3.2: A comparison of the absolute error of a 3-input addition between implementing with two 2-input adders and one 3-input adder

3.2 Fused Multiply-Add

FloPoCo's `DotProduct` unit is the only unit available that can be used to implement FMA. The design features a multiplier, with its non-truncated result fed into a large fixed-point accumulator. This design is ideal in a loop because the full result (that might possibly be very wide) can be stored in its entirety. The concept of a dot-product operator that internally uses fixed-point has also been described outside of FloPoCo by Kulisch in [21] as well as Lopes *et al.* in [22].

Figure 3.3 from Pasca [2] illustrates the parameters for the long accumulator. MaxMSB_X is the exponent of the highest most significant bit (MSB) of the inputs into the accumulator, MSB_A and LSB_A are the exponent of the most- and least-significant bit (LSB) of the accumulator. Parameter LSB_A effectively sets the accuracy of the accumulation as it determines the level of truncation. The condition $\text{MSB}_A \geq \text{MaxMSB}_X > \text{LSB}_A$ has to be true. Pasca [2, p. 121] notes that other implementations of a hybrid fixed- and floating-point accumulator in related work are more complex and tend to not as scale well with higher precisions.

In using FloPoCo's dot product unit it is observed that the sum has to be less than 2^{MSB_A}. The parameters are set as in equation set 3.4. The MSB's have one bit added on to hold the leading 1 in floating-point representation and an additional bit to allow 2's complement representation of negative numbers. These parameters assume continuous accumulation and hence the aim is to hold the possible sum in its entirety. MaxMSB_X depends on the maximum of the exponents to be added on which will either be the exponent of $(a \times b)$ or that of c.

$$\begin{aligned} \text{MaxMSB}_X &= 2 + \max\left({}^{\max}q_{a\times b},\ {}^{\max}q_c\right) \\ \text{MSB}_A &= \max\left(2 + {}^{\max}q_{(a\times b)+c}, \text{MaxMSB}_X\right) \\ \text{LSB}_A &= \min\left({}^{\min}q_{a\times b},\ {}^{\min}q_c\right) - w_F \end{aligned} \tag{3.4}$$

Assuming the FMA is for a single-use (without a continuous sum), the accumulator is set to

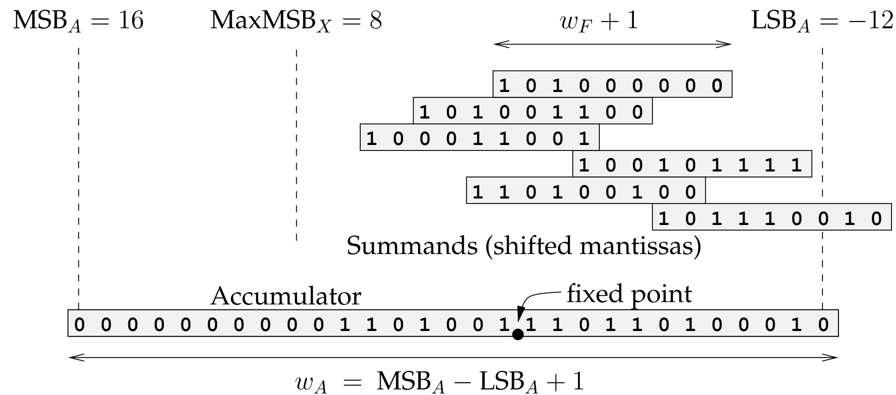


Figure 3.3: Parameters for FloPoCo's fixed-point accumulator, `LongAcc` [2]

accommodate only the larger of the two inputs when they're both at their smallest magnitude. In this worst-case, note that the accumulator has an extra bit at the end that is there to ensure the result has a correct least-significant bit, i.e. a carry from the lower bits and into the result's LSB is possible.

$$\mathrm{LSB}_A = \max\left({}^{\min}q_{a\times b},\ {}^{\min}q_c\right) - w_F - 1 \tag{3.5}$$

3.2.1 Area

To get an estimate of the number of LUTs for an FMA, its accumulator parameters have to be determined. After this, two FloPoCo units are generated, `DotProduct` (which includes `LongAcc`) and `LongAcc2FP`. The latter unit is needed to convert the accumulator's fixed-point representation back to floating-point.

The FMA with 3 parameters in addition to w_E and w_F presents a challenge in gathering area information. With the adders and multipliers, there are approximately $100 \times 10 = 1{,}000$ (w_F, w_E) possibilities for an operator and so far it has been practical to generate all possible units and synthesise them to get an estimate of their LUT counts. However, with the FMA's 3 additional parameters the number of possibilities is in the order of $2^8 \times 2^8 \times 2^8 = 16{,}777{,}216$ for single-precision $w_E = 8$ alone. It is not practical to generate and synthesise all parameters in this case so a new approach will be needed.

One possibility is generating and synthesising an FMA unit during runtime as needed. The way SOAP works would have to be changed to allow this as currently the area information is expected to already be stored internally. The implementation of this possibility should include a cache so that the number of LUTs gathered is stored and the same FMA unit would not have to be regenerated and synthesised in future.

An alternative possibility is to develop a heuristic model of the number of LUTs required. As Pasca [2] notes, the width of the accumulator is a key factor in determining how much resources

it uses. Therefore the 3 parameters can be reduced to 1 and bringing the number of possible FMA units to the order of $100 \times \sum_{w_E=5}^{15} (2^{w_E}) = 6,550,400$ (each exponent size sets the number of possible accumulator widths). This a great reduction but still too large and so more heuristics would be needed to reduce the w_E and w_F possibilities. For example, figure 3.1 revealed w_F to be the key parameter in determining the number of LUTs for a 3-input adder.

3.2.2 Error

The maximum absolute error for an operation $fma(a,b,c)$, which is $((a \times b)+c)$ in real arithmetic is expressed in equation 3.6. This model takes the maximum absolute error of performing a multiplication then an addition, but without the intermediate round-off error $\delta([a \times b])$.

$$\begin{aligned} &\epsilon\left(fma(a,b,c) \right) \\ &= (\epsilon(a) \cdot \epsilon(b) + \epsilon(a) \cdot \max(|b|) + \epsilon(b) \cdot \max(|a|)) + \epsilon([c]) + \delta([a \times b + c]) \end{aligned} \tag{3.6}$$

3.2.3 Transformations

To allow multiply-and-add expressions to be restructured to use the FMA unit, transformation rules are needed. These are illustrated in equation set 3.7. The commutativity rule in equation set 3.8 shows the only operand swap that is valid.

$$\begin{aligned} (a \times b) + c &\Longleftrightarrow fma(a,b,c) \\ a + (b \times c) &\Longleftrightarrow fma(b,c,a) \end{aligned} \tag{3.7}$$

$$fma(a,b,c) \Longleftrightarrow fma(b,a,c) \tag{3.8}$$

3.3 Constant Multiplier

FloPoCo has the floating-point constant multiplier `FPConstMult` described by Brisebarre *et al.* in [23]. They describe their fused unit to contain a simplified rounding stage that is no longer on the critical path, as is the case in a generic floating-point multiplier. In the generic multiplier, the product of the two $[1,2)$ significands is in $[1,4)$ but in a constant multiplier, the range can be known beforehand.

Furthermore, they prove that the result is always correctly rounded for all input floating-point numbers that are finite, periodic or otherwise infinitely representable (irrational). The shift-and-add approach is the core of multiplication by significands that are finitely representable in the particular w_F. Otherwise, at generation, the number of bits needed to represent the constant for a correctly rounded result is determined.

The developers of FloPoCo's constant multiplier note that previous work has typically assumed doubling of the mantissa size of irrational constants like π to achieve correct rounding. They demonstrate that the actual size needed can be computed using a continued fractions algorithm. In the worst case of doubling the mantissa size, the resource cost of correct rounding should be less than a factor of 2 as multiplication cost is sub-linear in the mantissa size of the constant.

3.3.1 Area

FloPoCo takes the constant as a string. The string argument can be an expression describing the constant, for example `"cos(e)"`, and FloPoCo acknowledges the mathematical constant $e \approx 2.718$ and calculates the cosine of the number. This is very ideal as no information about the constant is lost. However, SOAP would have to be extensively modified to allow representation, propagation and manipulation of these constant expressions. In this project, only constant numbers and not constant expressions are used. An assumption is made that the constant is finite in its decimal representation when passed as a string from SOAP to FloPoCo.

There is an infinite number of constants and hence impossible to synthesise all possibilities and store area information. Therefore, like the FMA, runtime area fetching is needed. Heuristics could be developed as an alternative by developing an area model based on the mantissa size needed for the constant as well as the number of shift-and-adds.

3.3.2 Error

The notation $constMult(v, a)$ is taken to represent $(v \times a)$ in real arithmetic, with v being a real constant value. The maximum absolute error for this operation is expressed in equation 3.9. This model takes the maximum absolute error of performing a multiplication, but assumes there is no round-off error in the value v, i.e. $\epsilon(v) = \delta(v) = 0$. The maximum absolute error has now eliminated two terms when compared to the that when using a generic multiplier.

$$
\begin{aligned}
&\epsilon(\ constMult(v, a)\ ) \\
&= \epsilon(a) \cdot |v| + \delta([v \times a])
\end{aligned} \tag{3.9}
$$

3.3.3 Transformations

To allow constant-multiplication expressions to be restructured to use the constant multiplier, transformation rules need to be added to those in SOAP. These are illustrated in equation set 3.10, where v is a real constant value. This operation has no rules for commutativity.

$$
\begin{aligned}
v \times a &\iff constMult(v, a) \\
a \times v &\iff constMult(v, a)
\end{aligned} \tag{3.10}
$$

Chapter 4

Implementation

The main interface in SOAP is through the `AreaErrorAnalysis` class which accepts an expression, input intervals and a precision. It has member method `frontier` for obtaining a set of Pareto-optimal expressions with their respective area and error information. The core of the class is another class, `TreeTransformer`, which uses transformation as well as reduction rules covered in section 2.2.3 to recursively gather a set of equivalent expressions.

An expression is represented using the class `Expr` that includes the methods `area` and `error`. SOAP had previously made the assumption that an expression would have 2 operands but this was changed throughout the program to allow any number of operands. Member method `area` calculates the total number of LUTs by counting unique uses of each operator (as described in section 2.2.1) and multiplying by the number of LUTs required for the operator.

The member method `error` of `Expr` is equivalent to $\epsilon : \textbf{Expression} \rightarrow \mathbb{R}$, which was described earlier in section 2.2.2 to gather the maximum absolute error of an expression. The function returns an instance of `ErrorSemantics` which holds the floating-point bounds (`FloatInterval`) for the result of the expression as well as the bounds for absolute error (`FractionInterval`). Class `FloatInterval` uses the `mpfr` data type from the *gmpy2* Python module to represent floating-point numbers of a particular precision as well as perform arithmetic that emulates floating-point operations like addition and multiplication with a specified precision context. `FractionInterval` uses `mpq` data type, also from *gmpy2*, to represent real rational numbers and perform their arithmetic.

The cache of area information mentioned throughout chapter 3 is implemented as a Python dictionary. The key to this mapping is an ordered collection of the operator and its parameters. The dictionary is stored into a file as a compact byte stream by using the Python module *pickle*. This form is much more efficient than other methods such as JavaScript Object Notation (JSON) which stores a human-readable text representation of the data. The dictionary is loaded at runtime, or

created if not present, and updated after any cache miss. The cache is primarily for the FMA and the constant multiplier which have been shown in chapter 3 to be impractical to pre-compute the area information of all their possibilities. A parameter has been adder in `soap.common` to determine whether or not to assume single or multiple use of the FMA.

4.1 Adding Operators

To add a new operator, firstly the syntax was defined, e.g. `add3`, `constMult`. Next, the operator's properties of associativity and distributivity are defined by placing it into the respective property groups. For commutativity, this involves adding a condition for the operator in the recursive walk of `TreeTransformer` and defining which operands are commutable.

For the 3-input adder, area information was gathered and stored by sweeping through all possible values of w_E and w_E (as seen in figure 3.1). For the FMA and the constant multiplier, the sequence of FloPoCo commands to be run and their arguments are defined.

The next step is to add a condition in `ErrorSemantics.do_op` function to define propagation of error for the new operator. This function evaluates the interval of absolute error after an operation as well as the floating-point interval for the result. Transformation methods that describe equivalence rules for the operator are added to a child class of `TreeTransformer`.

Chapter 5

Testing

In this chapter, the functional correctness of what has been implemented is established. Equations are evaluated to obtain exact expected results and compared to determine what works.

The results that follow and throughout this report include time durations of the analysis but they are primarily given to illustrate the relative magnitudes. SOAP's internal runtime caches—and excluding the area information caches—were invalidated before each timed analysis. The ideal timing conditions would be to run the program on idle machines such as the remote compute units on Amazon Web Services (AWS). However, while still in development and requiring external software such as Xilinx's ISE Design Suite to be installed, a standard laptop was used as the testing platform.

Singular frontiers with only one point are plotted with non-optimal points to illustrate transformations.

5.1 Three-Input Adder

To begin with, the simple expression in equation 5.1 is used, where one might know beforehand that the input variables have the ranges shown in equation set 5.2.

$$a + b + c \tag{5.1}$$

$$\begin{aligned} a &\in [1, 100] \\ b &\in [0.01, 1] \\ c &\in [0.1, 10] \end{aligned} \tag{5.2}$$

What should be noted about these intervals is that that with the largest magnitude is written first in the expression and as explained earlier in section 2.1.1, this order can degrade the numerical

accuracy if intermediate results are rounded. The above parameters were run for single precision ($w_F = 23$) to obtain the frontiers in figure 5.1.

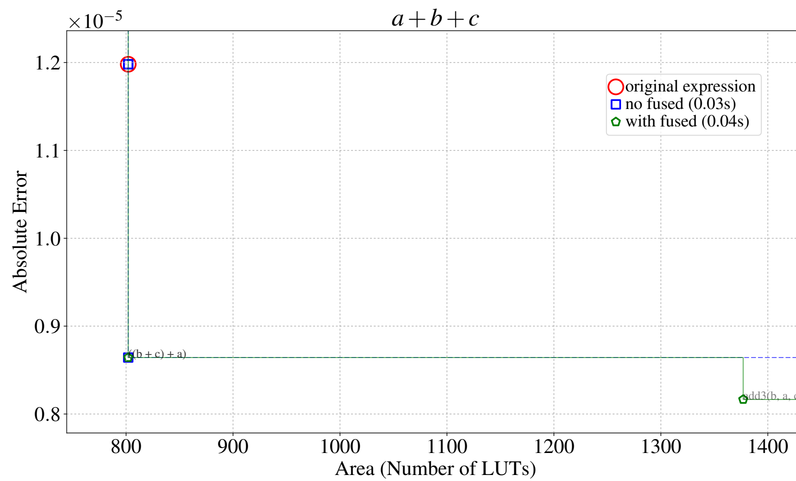


Figure 5.1: Frontiers before and after including the 3-input adder for 3-operand addition at single precision

First to note is that the original expression is not optimal in either one of the two frontiers. The original $((a+b)+c)$ straightforward use of the 2-input adder is optimised to $((b+c)+a)$; the latter adds numbers in order of increasing worst-case absolute error which is evident by the increasing maximum magnitude. Secondly, the 3-input adder improves the error but marginally compared with through 2-input associativity and at a significant area cost ($1.7\times$).

Correctness will be tested by first evaluating the area. The combined interval for the input and output values is $[0.01, 111]$. This range falls within the range $[0.0078125, 128)$ of binary powers and hence the exponent range required is $[-7, 6]$. This exponent range would need an exponent size w_E of 4 but this is taken up to the minimum synthesisable exponent size of 5.

A synthesised 2-input adder with parameters $w_E = 5$ and $w_F = 23$ uses 401 LUTs and a 3-input adder with the same parameters uses 1377 LUTs. This results is 802 LUTs when two 2-input adders are used. All of this is consistent with figure 5.1.

The error of the original expression $((a+b)+c)$ is given in equation 5.3, where ϵ gives the maximum absolute error of an expression and δ gives the maximum round-off error for an interval.

ϵ : **Expression** $\to \mathbb{R}$ and δ : **Interval**$_{\mathbb{F}} \to \mathbb{R}$.

$$
\begin{aligned}
\epsilon((a+b)+c) &= \epsilon(a+b)+\epsilon(c)+\delta([(a+b)+c]) \\
&= \delta([a])+\delta([b])+\delta([a+b])+\delta([c])+\delta([(a+b)+c]) \\
&= \delta([1,100])+\delta([0.01,1])+\delta([1.01,101])+\delta([0.1,10])+\delta([1.11,111]) \\
&= \frac{1}{2}\cdot 2^{-23}\cdot(64+1+64+8+64) \\
&\approx 1.1981\times 10^{-5}
\end{aligned} \tag{5.3}
$$

Note that $\delta([(a+b)+c])$ is used as a shorthand for $\delta([fl(a+b)+c])$. The calculated maximum absolute error in equation 5.3 matches exactly that of both frontiers in figure 5.1. Continuing on, the same is done for the more optimal rearrangement $((b+a)+c)$ in equation 5.4 and the fused unit $add3(a,b,c)$ in equation 5.5 to prove correctness.

$$
\begin{aligned}
\epsilon((b+c)+a) &= \epsilon(b+c)+\epsilon(a)+\delta([(b+c)+a]) \\
&= [\delta([b])+\delta([c])+\delta([b+c])]+\delta([a])+\delta([(b+c)+a]) \\
&= \delta([0.01,1])+\delta([0.1,10])+\delta([0.11,11])+\delta([1,100])+\delta([1.11,111]) \\
&= \frac{1}{2}\cdot 2^{-23}\cdot(64+1+64+8+64) \\
&\approx 8.6427\times 10^{-6}
\end{aligned} \tag{5.4}
$$

$$
\begin{aligned}
\epsilon(add3(a,b,c)) &= \epsilon(a)+\epsilon(b)+\epsilon(c)+\delta([a+b+c]) \\
&= \delta([a])+\delta([b])+\delta([c])+\delta([a+b+c]) \\
&= \delta([1,100])+\delta([0.01,1])+\delta([0.1,10])+\delta([1.11,111]) \\
&= \frac{1}{2}\cdot 2^{-23}\cdot(64+1+8+64) \\
&\approx 8.1658\times 10^{-6}
\end{aligned} \tag{5.5}
$$

Note that the difference between equations 5.4 and equations 5.5 and hence the error reduction of the 3-input adder is the round-off error of the best-case intermediate result $(b+c)$.

In version 1 of SOAP, it is assumed that the input variables are intervals of real numbers and hence they have an associated error when represented as a binary floating-point. In version 3 however, a user can set the error interval of the inputs and the default is zero. This means that zero error at the inputs, i.e. the floating-point value of an input matches its real value, results in $\delta([a])=\delta([b])=\delta([c])=0$. This in turn results in the simplified equation of the error improvement of the 3-input adder as given by equation 5.6.

$$
\frac{\epsilon((b+c)+a)}{\epsilon(add3(a,b,c))} = \frac{\delta([b+c])+\delta([(b+c)+a])}{\delta([a+b+c])} \tag{5.6}
$$

If $|a+b+c| \ll |b+c|$ then the error improvement can be arbitrarily large since $\delta([a+b+c])$ will be much smaller than the numerator in equation 5.6. Therefore in general, summations with intermediate results that are far greater in magnitude than that of the final result will benefit greatly in numerical accuracy through the use of multiple-input adders.

5.2 Fused Multiply-Add

For, the simple expression in equation 5.7 is used with input variables having the ranges shown in equation set 5.8.

$$(a \times b) + c \tag{5.7}$$

$$a \in [1, 100], \quad b \in [0.01, 1], \quad c \in [0.1, 10] \tag{5.8}$$

Figure 5.2 shows the resulting frontiers. First to note is that the FMA unit has improved on both area and error of the original expression. The minimum synthesisable exponent size of 5 covers the exponent range needed for the multiplication and for the addition. A multiplier and an adder, both of $w_E = 5$ and $w_F = 23$, require $121 + 401 = 522$ LUTs.

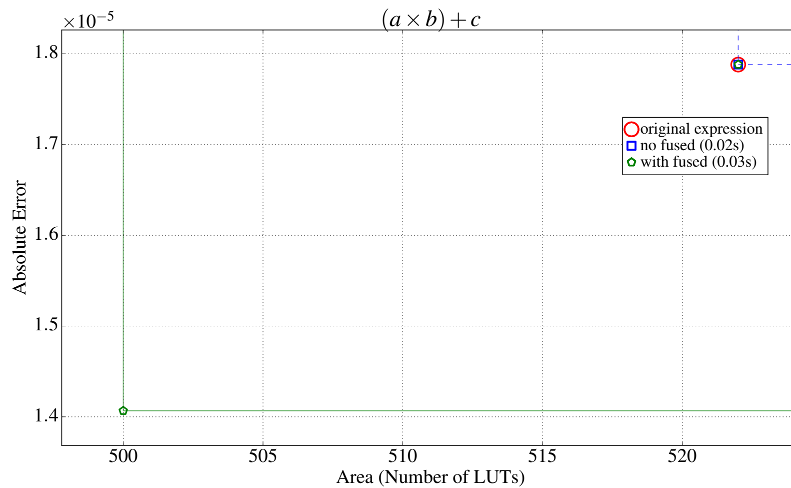


Figure 5.2: Frontiers before and after including multiple-use FMA for a simple expression at single precision

To determine parameters for the FMA, the exponent bounds for the input into the accumulator are evaluated. The exponent range needed for $(a \times b) \in [0.01, 100]$ is $[-7, 6]$, for $c \in [0.1, 10]$ is $[-4, 3]$ and for the result $((a \times b) + c) \in [0.11, 110]$ is $[-4, 6]$. MaxMSB_X is the maximum of the accumulator inputs' exponent upper bounds, 6, with 2 bits added on as described earlier in equation 3.4. MSB_A is maximum out of MaxMSB_X and the output's exponent upper bound ,6, with 2 bits added on. LSB_A is taken to be such that the accumulator input with the lowest exponent can be stored entirely without any loss, $(-7 - w_F)$. An FMA—that includes conversion from the accumulator's fixed-point representation back to floating-point—with parameters $\text{MaxMSB}_X = 8$, $\text{MSB}_A = 8$ and $\text{LSB}_A = -30$ uses $309 + 191 = 500$ LUTs.

The maximum absolute errors for the original and for the fused expressions are included in equations 5.9 and 5.10; they are both consistent with the plot in figure 5.2.

$$
\begin{aligned}
\epsilon((a \times b) + c) &= \epsilon(a \times b) + \epsilon(c) + \delta([(a \times b) + c]) \\
&= (\epsilon(a) \cdot \epsilon(b) + \max(|b|) \cdot \epsilon(a) + \max(|a|) \cdot \epsilon(b) + \delta([a \times b])) + \delta([c]) + \delta([(a \times b) + c]) \\
&= \delta([a]) \cdot \delta([b]) + \delta([a]) + 100 \cdot \delta([b]) + \delta([a \times b]) + \delta([c]) + \delta([(a \times b) + c]) \\
&= \delta([1,100]) \cdot \delta([0.01,1]) + \delta([1,100]) + 100 \cdot \delta([0.01,1]) + \delta([0.01,100]) + \delta([0.1,10]) + \delta([0.11,110]) \\
&= \frac{1}{2} \cdot 2^{-23} \cdot \left(\frac{1}{2} \cdot 2^{-23} \cdot 64 + 64 + 100 + 64 + 8 + 64 \right) \\
&\approx 1.7881 \times 10^{-5}
\end{aligned}
\tag{5.9}
$$

$$
\begin{aligned}
\epsilon(fma(a,b,c)) &= \epsilon(a \times b) + \epsilon(c) + \delta([(a \times b) + c]) \\
&= (\epsilon(a) \cdot \epsilon(b) + \max(|b|) \cdot \epsilon(a) + \max(|a|) \cdot \epsilon(b)) + \delta([c]) + \delta([a \times b + c]) \\
&= \delta([a]) \cdot \delta([b]) + \delta([a]) + 100 \cdot \delta([b]) + \delta([c]) + \delta([(a \times b) + c]) \\
&= \delta([1,100]) \cdot \delta([0.01,1]) + \delta([1,100]) + 100 \cdot \delta([0.01,1]) + \delta([0.1,10]) + \delta([0.11,110]) \\
&= \frac{1}{2} \cdot 2^{-23} \cdot \left(\frac{1}{2} \cdot 2^{-23} \cdot 64 + 64 + 100 + 8 + 64 \right) \\
&\approx 1.4067 \times 10^{-5}
\end{aligned}
\tag{5.10}
$$

If the floating-point representation of the input variables are taken to be exact and hence $\epsilon(a) = \epsilon(b) = \epsilon(c) = 0$, then error improvement is given by equation 5.11. Similarly to the 3-input adder, it is evident that if $|a \times b| \gg |(a \times b) + c|$, i.e. if the intermediate result is far greater in magnitude than the final result, then an arbitrarily large error improvement is achieved.

$$
\frac{\epsilon((a \times b) + c)}{\epsilon(fma(a,b,c))} = \frac{\delta([a \times b]) + \delta([(a \times b) + c])}{\delta([a \times b + c])}
\tag{5.11}
$$

5.2.1 The Single-Use FMA

The previous analysis of the FMA was assuming the accumulator size was made to hold the exact result in its entirety, orientated towards continuous accumulation such as that as part of a loop. Here, the single-use design of an FMA is used where the accumulator size is such that it is minimum in size to allow correct accumulation of only 2 summands. The frontiers are displayed in figure 5.3.

Firstly, the absolute errors are exactly the same as before because the errors represent that of a single iteration of the expression. The parameters MaxMSB_X and MSB_A remain the same and LSB_A is the only one that changes. The minimum exponent needed for $(a \times b)$ is -7 and that for c is -4, therefore LSB_A is made to be $(-4 - w_F - 1)$ as per equation 3.5. An FMA—that includes conversion back to floating-point—with parameters $\text{MaxMSB}_X = 8$, $\text{MSB}_A = 8$ and $\text{LSB}_A = -28$ uses

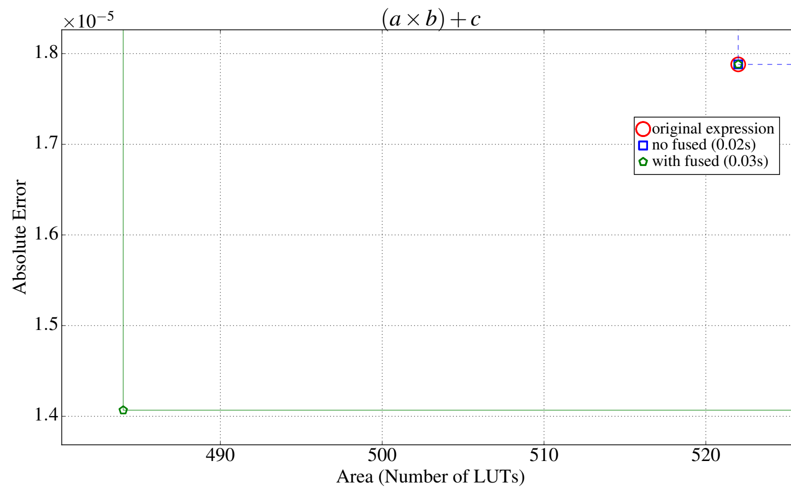


Figure 5.3: Frontiers before and after including single-use FMA for a simple expression at single precision

$297 + 187 = 484$ LUTs. The area improvement has now increased due to a smaller accumulator, from 1.04× assuming multiple uses to 1.08× with a single-use FMA. If the difference between the minimum exponent of $(a \times b)$ and that of c is larger then there will be a larger change in area improvement.

5.3 Constant Multiplier

Single-precision representation of π is only accurate to the first 7 decimal digits. If more digits are added as in equation 5.12, a generic multiplier will be unaware of them whereas the implemented constant multiplier should be generated such that there is correct rounding of the result as if there was no round-off error in the constant. Figure 5.4 shows the resulting frontiers.

$$\begin{aligned} &a \times 3.14159265358979323846264338327\,95 \\ &a \in [-1, 1] \end{aligned} \tag{5.12}$$

The number of LUTs needed to synthesise a $(w_E = 8, w_F = 23)$ multiplier by the longer π approximation is 342, which matches the analysis output. A generic multiplier of similar parameters uses 136 LUTs and so the area cost of the constant multiplier is 2.51× which is more than what was previously suggested in the publication in section 3.3.

Note that SOAP calculates the exact round-off error of a constant number: $\delta(3.14\ldots) = (3.14\ldots) - fl(3.14\ldots) \approx -8.7423 \times 10^{-8}$, where fl is the mapping $\mathbb{R} \to \mathbb{F}$, with $w_E = 8$ and $w_F = 23$. Proofs for the maximum absolute errors of the original expression and the constant multiplier implementation are given in equations 5.13 and 5.14. For the constant multiplier, the assumption is $\epsilon(3.14\ldots) = \delta(3.14\ldots) = 0$, i.e. there is no round-off error in the constant. Absolute error has improved 1.28× with

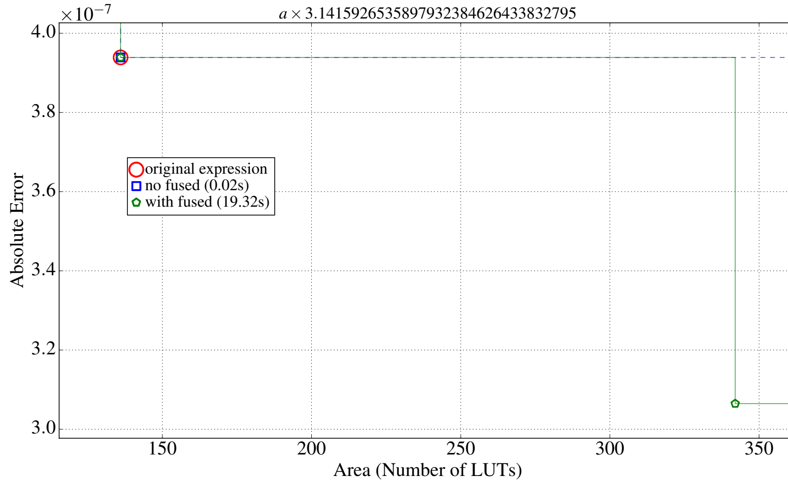


Figure 5.4: Frontiers before and after including the constant multiplier, for a π multiplier at single precision input/output

the constant multiplier.

$$\begin{aligned}
\epsilon(a \times 3.14\ldots) &= (\epsilon(a)\cdot\epsilon(3.14\ldots) + |3.14\ldots|\cdot\epsilon(a) + \max(|a|)\cdot\epsilon(3.14\ldots) + \delta([a\times 3.14\ldots])) \\
&= (\delta([a])\cdot\delta(3.14\ldots) + (3.14\ldots)\cdot\delta([a]) + \max(|a|)\cdot\delta(3.14\ldots) + \delta([a\times 3.14\ldots])) \\
&= (\delta([-1,1])\cdot\delta(3.14\ldots) + (3.14\ldots)\cdot\delta([-1,1]) + 1\cdot\delta(3.14\ldots) + \delta([-3.14\ldots,3.14\ldots])) \\
&= \frac{1}{2}\cdot 2^{-23}\cdot\left(\frac{1}{2}\cdot 2^{-23}\cdot 2 + (3.14\ldots) + 2\cdot((3.14\ldots) - fl(3.14\ldots))\right) \\
&\approx 3.9389\times 10^{-7}
\end{aligned} \tag{5.13}$$

$$\begin{aligned}
\epsilon(\,constMult(3.14\ldots,a)\,) &= \epsilon(a)\cdot|3.14\ldots| + \delta([a\times 3.14\ldots]) \\
&= \delta([-1,1])\cdot(3.14\ldots) + \delta([-3.14\ldots,3.14\ldots]) \\
&= \frac{1}{2}\cdot 2^{-23}\cdot((3.14\ldots)+2) \\
&\approx 3.0646\times 10^{-7}
\end{aligned} \tag{5.14}$$

Chapter 6

Results

In this chapter, the implemented program is quantitatively evaluated to investigate the extent of improvements made. The inner expressions of PolyBench [3] and Livermore Loops [24] are used. These two benchmark suites contain kernels that are widely used to evaluate parallel computers. In this project, 17 benchmark expressions that are nested inside a selection of the kernels are used. These chosen kernels range in applications from fluid dynamics to statistics.

Listing 2 shows a snippet of the `2mm` benchmark kernel from PolyBench which performs 2 matrix multiplications. The triple-nested highlighted instructions in lines 94 and 101 are transformed to the expressions in equations 6.1 and 6.2 respectively. They are then labelled in this project as `2mm_1` and `2mm_2` respectively. The variable `alpha` is constant during runtime, with a default value of 1.5. To obtain input intervals for PolyBench kernels, the array initialisation functions were studied and for Livermore Loops a range of $[0,1]$ is assumed. All the benchmark expressions and respective input intervals are listed in Appendix B.

$$t + (alpha \times a \times b) \xrightarrow{alpha=1.5} t + (1.5 \times a \times b) \tag{6.1}$$

$$d + (t \times c) \tag{6.2}$$

6.1 Single-Precision, Multiple-Use FMA

The 17 benchmark expressions were run at single-precision ($w_F = 23$) and assuming multiple use of the FMA to obtain the result summary in table 6.1. The analysis times in the table are such that there was always a hit in the area dynamic cache, i.e. a first time run to fetch and store the area information then run a second time.

```
87  #pragma scop
88    /* D := alpha*A*B*C + beta*D */
89    for (i = 0; i < _PB_NI; i++)
90      for (j = 0; j < _PB_NJ; j++)
91        {
92          tmp[i][j] = SCALAR_VAL(0.0);
93          for (k = 0; k < _PB_NK; ++k)
94            tmp[i][j] += alpha * A[i][k] * B[k][j];
95        }
96    for (i = 0; i < _PB_NI; i++)
97      for (j = 0; j < _PB_NL; j++)
98        {
99          D[i][j] *= beta;
100         for (k = 0; k < _PB_NJ; ++k)
101           D[i][j] += tmp[i][k] * C[k][j];
102       }
103 #pragma endscop
```

Listing 2: Snippet from the 2mm kernel in PolyBench [3]

Area Improvement compares the points of least area on the Pareto frontiers: $\frac{\text{original best area}}{\text{new best area}}$ and similarly for error. *Area Cost* compares the area needed to achieve the best error on the frontiers: $\frac{\text{area of new best error}}{\text{area of original best error}}$. And finally, *Analysis Time* is $\frac{\text{new time}}{\text{old time}}$.

Looking at table 6.1, it is immediately evident that the implemented program in this project extends version 1 of SOAP without losing previous optimal points, as can be seen by the fact that all area and error improvements are greater than or equal to one. Secondly, the geometric means reveal overall noticeable improvements: 1.036× in area and 1.187× in error at an area cost of 1.793×. Overall, there is a modest increase in analysis time that surges up for compounded expressions like `state_frag` and `heat-3d`.

Individual cases will now be analysed to further explain the results of table 6.1. As previously, singular frontiers are expanded with non-optimal points to illustrate transformations.

The best area improvement of 1.131× was achieved in `fdtd-2d_1` and figure 6.1 shows its frontiers. The equivalent expression with the best area is illustrated in equation 6.3. The constant 0.5 is an exponent of 2 and so only requires one shift, making the constant multiplier very minimum in digital logic. Some error improvement was obtained by the transformed expression $fma(0.5, (b+c), a))$.

$$\begin{array}{c} a + (0.5 \times (c+b)) \\ \downarrow \\ a + constMult(0.5,\ (b+c)) \end{array} \tag{6.3}$$

The best error improvement of 1.433× was achieved in `jacobi-1d` and figure 6.2 shows its frontiers. The equivalent expression with the best error is illustrated in equation 6.5. It should be noted that the number 0.33333 is exactly as it appears in the PolyBench source code. The transformation with the best error was due to the use of two fused units, a constant multiplier and a 3-input adder. Note that one of the two new mid-frontier expressions is $constMult(0.33333, ((b+c)+a))$ which does

Benchmark Name	Area Improvement	Error Improvement	Area Cost	Analysis Time
`2d_hydro`	1.000×	1.270×	1.742×	10.7×
`2mm_1`	1.115×	1.000×	2.102×	1.5×
`2mm_2`	1.000×	1.000×	2.544×	2.1×
`3mm`	1.000×	1.000×	2.443×	1.3×
`correlation`	1.000×	1.254×	1.880×	1.0×
`deriche`	1.000×	1.176×	1.666×	3.3×
`fdtd-2d`	1.000×	1.157×	1.541×	1.4×
`fdtd-2d_1`	1.131×	1.143×	1.791×	2.0×
`gemm`	1.000×	1.168×	2.219×	1.0×
`gemver`	1.000×	1.227×	1.775×	1.3×
`heat-3d`	1.053×	1.286×	1.518×	66.2×
`jacobi-1d`	1.000×	1.433×	1.551×	1.2×
`seidel`	1.000×	1.313×	1.374×	4.7×
`state_frag`	1.000×	1.294×	2.170×	190.1×
`symm`	1.115×	1.283×	1.532×	8.2×
`syr2k`	1.115×	1.133×	1.198×	6.4×
`syrk`	1.115×	1.143×	2.051×	1.2×
Min	**1.000**×	**1.000**×	**1.198**×	**1.0**×
Max	**1.131**×	**1.433**×	**2.544**×	**190.1**×
Geomean	**1.036**×	**1.187**×	**1.793**×	**3.5**×

Table 6.1: Summary of benchmark results for single precision and with the multiple-use implementation of the FMA

not improve the area of the original expression because the number does not require a small number of shift-and-add operations.

$$0.33333 \times ((a+b)+c)) \\ \downarrow \\ \mathit{constMult}(0.33333,\ \mathit{add3}(a,b,c))) \tag{6.4}$$

The worst area cost to achieve the best error was 2.544× in `2mm_2` and figure 6.3 shows its frontiers. The culprit equivalent expression is the only fused arithmetic expression on the frontier—since the expression is quite simple. Equation 6.5 illustrates the transformation to the FMA unit. The exponent ranges needed for the FMA's accumulator, assuming multiple uses, are: $q_d \in [-126, 22]$, $q_{t\times c} \in [-126, 0]$, $q_{d+(t\times c)} \in [-126, 22]$. This results in a very wide accumulator with a width of 173 bits. This arises from the fact that at least one of the input intervals includes zero and hence the exponent lower bound is the minimum exponent value for the particular exponent size: ${}^{\min}q = -(2^{w_E-1}-2) = -126$.

$$d + (t \times c)) \\ \downarrow \\ \mathit{fma}(t,\ c,\ d) \tag{6.5}$$

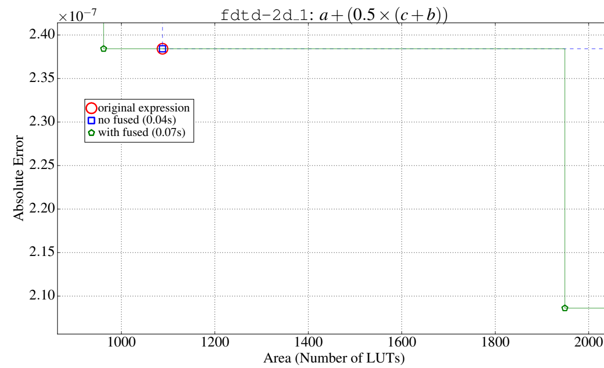


Figure 6.1: Frontiers for `fdtd-2d_1` at single precision, and with the multiple-use FMA

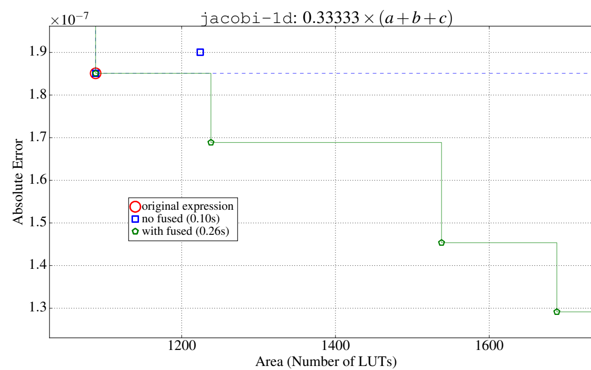


Figure 6.2: Frontiers for `jacobi-1d` at single precision, and with the multiple-use FMA

The worst analysis time was 190.1× in `state_frag` and figure 6.4 shows its frontier alongside non-Pareto optimal equivalent expressions. The increased analysis time is explained by the increase in the number of equivalent expressions to analyse. At a transformation depth of 3, the fused frontier has 69× the number of equivalent expressions to analyse. The transformed expression with the best error has 8 FMA units, achieving an error improvement of 1.294× at an area cost of up to 2.170× more LUTs. Similarly to the previous case of worst area cost, the FMA units here have very wide accumulators due to accumulator input ranges including zero and hence the accumulator's LSB exponent equalling ${}^{\min}q$.

When SOAP's greedy trace (explained in section 2.2.3) is used instead of performing a full closure, the frontiers in figure 6.5 are obtained. With this search algorithm, the area and error improvements are the same as well as the area cost but now at a much reduced analysis time of 9.8×, down from 190.1×. With the greedy trace, the fused frontier has reduced the number of expressions to analyse by a factor of 11. Therefore, taking this extreme benchmark, the greedy trace is seen to improve the analysis time while still providing the same optimal expressions.

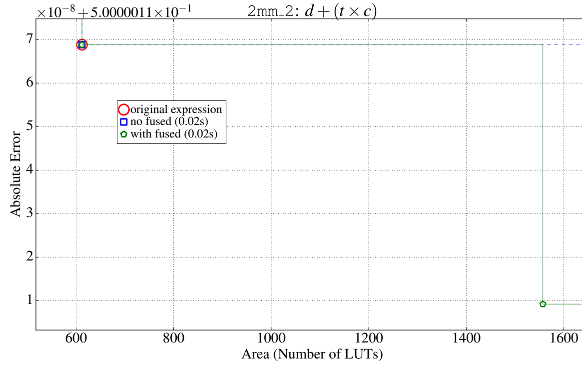


Figure 6.3: Frontiers for 2mm_2 at single precision, and with the multiple-use FMA

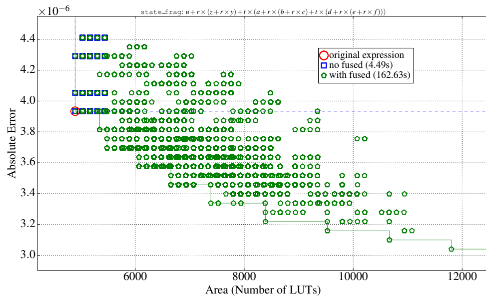


Figure 6.4: Full closure

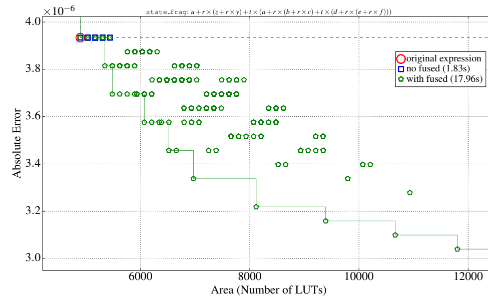


Figure 6.5: Greedy trace

Figure 6.6: Frontiers for `state_frag` at a transformation depth of 3, at single precision and with the multiple-use FMA

6.2 Single- and Double-Precision, Single- and Multiple-Use FMA

Analysis very similar to the preceding section was performed for single precision ($w_F = 23$) and double precision ($w_F = 52$), as well as either assuming single or multiple uses of the FMA. The aggregated results are included in table 6.2.

First to note is that there is an almost negligible difference in improvements as well as area cost for the benchmark expressions between the single- and the multiple-use FMA. This is attributed to the fact that every interval of all benchmark expressions includes zero and hence the FMA's accumulator has to be very wide even when assuming single use.

Secondly, at double precision the worst area cost increased significantly compared to at single precision. This is due to a large standard exponent size ${}^{\text{double}}w_E = 11$, compared to ${}^{\text{single}}w_E = 8$. The FMA in 2mm_2 has an $\text{LSB}_A = {}^{\min}q = -(2^{w_E - 1} + 2)$ and hence an $8\times$ increase in its accumulator width from single to double precision. Thirdly, the trends and reasoning in the preceding section also apply to double precision as seen by the recurring extreme benchmarks for each respective frontier

Precision	FMA Type	Best Area Improvement	Best Error Improvement	Worst Area Cost	Worst Analysis Time
single	single-use	1.13× (fdtd-2d_1)	1.43× (jacobi-1d)	2.55× (2mm_2)	190× (state_frag)
	multiple-use	1.13× (fdtd-2d_1)	1.43× (jacobi-1d)	2.54× (2mm_2)	160× (state_frag)
double	single-use	1.33× (fdtd-2d_1)	1.31× (jacobi-1d)	7.87× (2mm_2)	120× (state_frag)
	multiple-use	1.34× (fdtd-2d_1)	1.31× (jacobi-1d)	7.87× (2mm_2)	130× (state_frag)

Table 6.2: Aggregate of benchmark results for single and double precision, and for the two types of FMA units

measurement.

Fourthly, the best area improvement increases with a higher precision due to the constant multiplier in `fdtd-2d_1`. The constant is 0.5 and so the number of LUTs needed to implement the multiplier is fairly unchanging in w_F compared to implementing with a generic floating-point multiplier.

Fifth to note is that the best error improvement decreases with increasing precision because the fused units primarily boost the accuracy by reducing intermediate rounding-off errors. With a higher precision, the rounding-off error decreases at an exponential rate as evident in the unit-of-last-place definition in equation 2.7.

Chapter 7

Evaluation

In chapter 6, the fused-arithmetic frontiers of benchmark expressions were compared to the original frontiers using generic discrete-arithmetic units and seen to make significant improvements. In this chapter, the fused frontier is evaluated against a common method of varying precision, as well as version 3 of SOAP which considers latency and analyses full numerical programs.

7.1 Against Varying Precision

A commonplace method to improve numerical accuracy is to increase the precision of operators in the datapath. The `seidel` benchmark is used to make a comparison for single precision and assuming multiple-use of the FMA. As figure 7.1 shows and seen in table 6.1, the best error improves by 1.3× and the best area is exactly the same as that using discrete arithmetic units. However, when the precision of the frontier points is varied, it is evident that at some precisions there is both an area and error improvement. Between single and double precision, the best area improvement now becomes 1.04×.

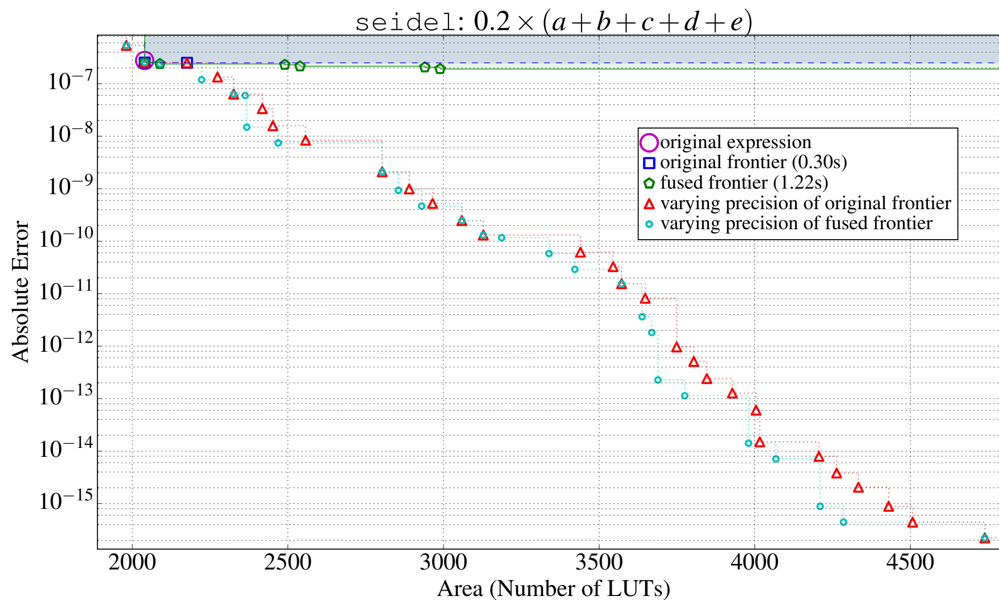


Figure 7.1: Frontiers for varying the precision of the frontier of `seidel` at single precision, and with the multiple-use FMA

7.2 Against SOAP Version 3

Unfortunately, direct comparisons of area in number of LUTs between versions 1 and 3 of SOAP are not very meaningful for a number of reasons. Firstly, the target device is different—Virtex 7 instead of Virtex 6 in version 1—so the number of LUTs is not directly comparable although indicative. Secondly, SOAP 3 drops the ability to synthesise operators of specific as-needed exponent size and precision, and instead only has area information for IEEE-754 standard single and double precision for each operator. Thirdly, version 3 considers latency of programs and so it uses its scheduling ability to reduce the number of units by pipelining them, whereas version 1 in comparison calculates the area for combinatorial logic.

With the above taken into account, an attempt is made to compare frontiers between the two different versions of SOAP. The `seidel` benchmark was used and version 1 of SOAP was modified to count operators in the case that the number of operators was reduced. For example, the original expression $0.2 \times (a + b + c + d + e)$ is taken to require one multiplier and one one (pipelined) adder. The input error intervals given into version 3 of SOAP are such that they match those modelled in version 1.

The result of this evaluation is figure 7.2 where area should be ideally ignored but the proximity between the versions is seen to be significant. It should also be noted that version 3 of SOAP includes underflow error in its estimation of the absolute error, and hence the sightly larger error values in its frontier. Also to note is that the frontier of SOAP 3 is three-dimensional (latency being the third) and

hence the presence of the point that does not appear to be Pareto-optimal in this view.

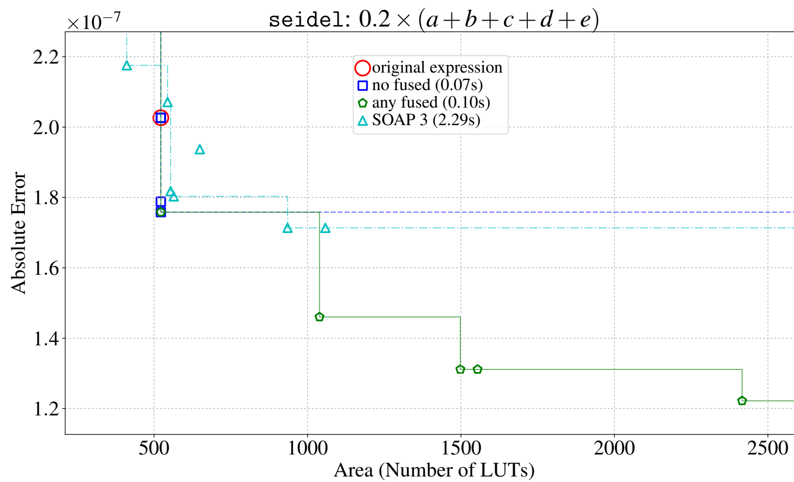


Figure 7.2: Frontiers for `seidel` at single precision, and with the multiple-use FMA, compared to a matched and modified version 3 of SOAP

What is evident from figure 7.2 is the significant error improvement made by the constant multiplier and the 3-input adder, even when compared to version 3 of SOAP (1.4× improvement). This illustrates that if fused arithmetic units were implemented into version 3 of SOAP, then the frontiers have potential to expand into greater optimality.

Chapter 8

Conclusions

This project has successfully shown what is possible with fused arithmetic units and improvements that can be made in a high-level synthesis flow. Furthermore, the transformation capabilities of SOAP are harnessed to analyse areas of optimality in area and numerical accuracy. In the process, this project has presented challenges with implementing fused arithmetic units and solutions such as runtime area fetching while making use of a dynamic cache.

While there was significant analysis time increase with large expressions, it was demonstrated in section 6.1 that using SOAP's greedy trace reduced this increase significantly while still obtaining the same optimal equivalent expressions.

It has also been shown empirically in chapter 5 that the 3-input adder and the fused multiply-add can make arbitrarily large accuracy improvements if the magnitude of their intermediate results is far greater than that of their final result.

Version 3 of SOAP has not been able to reduce the resource usage of any of its 11 benchmark programs (from PolyBench and Livermore Loops) as 'they have no redundant computations' [9, p. 2]. However, this project has shown that noticeable area improvements can be made to the inner expressions of benchmark programs by fused arithmetic units, and specifically the constant multiplier. These fused units open up the possibilities for a tool evaluating a numerical program as a whole.

8.1 Further Work

Tighter Interval Bounds

Tracking the bounds of an expression's value using interval arithmetic is an overestimation and the margin grows with the number of operations. A tighter expression value interval results in a

better estimate of the maximum round-off error. The better and exact method of evaluating output value interval would be to perform minimisation and maximisation on the expression with the given intervals to determine exact bounds of the expression value. For example, the interval of $a \times (1-a) \times (1+a)$ when $a \in [0,2]$ evaluates to $[-1,1] \times [0,2] \times [0,2] = [-2,2]$ using interval arithmetic whereas the exact interval is approximately $[-0.58, 0.58]$ as obtained from solving two constrained optimisation problems on a third-order polynomial. Interval arithmetic, in this case, results in the round-off error to be an overestimation by a factor of 4. Python module *scipy* [25] seems promising for solving constrained optimisation problems.

FMA Accumulator Design Space

Firstly, as already mentioned in section 3.2.1, the implemented fetching of the FMA's area information during runtime has the alternative of using a heuristic model. Some analysis will be needed to reduce the parameters to 2. The floating-point multiplier that is built into the `DotProduct` should be analysed as was done in figure 3.1 for the 3-input adder to determine the key parameter.

Secondly, it was discovered in chapter 6 that all benchmarks—and hence most real-world applications—have input intervals that include zero. The accumulator in this project was made to cover the minimum exponent range, ${}^{\min}q$, at a cost. Moving on, SOAP's expression design space could be expanded to include varying accumulator widths by varying LSB_A as in equation 8.1 where the width increases in α multiples of w_F. This would require expanding the error model of the FMA to take into account non-ideal accumulators widths and their round-off errors due to any loss in least-significant bits.

$$\mathrm{LSB}_A = \mathrm{MSB}_A - (\alpha \times w_F) - 1 \tag{8.1}$$

SOAP Version 3 and Latency

Limiting the scope of this project to area and accuracy trade-offs has set the foundation to expand the work with fused arithmetic units towards considering latency and full programs in version 3 of SOAP. Fused units with shorter critical paths are expected to reduce latency and those that reduced error are expected to make larger improvements as part of a loop. Area improvement of loop content is expected to scale with the level of loop unrolling.

The FMA has been implemented with loops in mind through the multiple-use type in this project. The constant multiplier in this project has potential to be used to implement division by constants. Furthermore, version 3 of SOAP could explore the design space of implementing division by small integers by using FloPoCo's `FPConstMultRational` unit as De Dinechin *et al.* demonstrate in [26].

Appendix A: User Guide

The project's source code, including that used to generate graphs, is available to download from `https://github.com/eugenius1/soap`. This guide is also available in the `README.md` file in the repository.

A.1 Installing

These instructions are given for Ubuntu and are expected to work for other major Linux operating systems.

1. Download or clone the git repository. The default branch should be `eusebius/soap1/fused`.

2. Install matplotlib:
   `http://matplotlib.org/users/installing.html#build-requirements`

3. Install Python3 if not already installed:
   `https://www.python.org`

4. Install gmpy2. From the shell:
   `sudo apt-get install python3-gmpy2`

5. Install other Python dependencies. From the project directory:
   `pip3 install -r requirements.txt`

Optionally, in order to allow fetching of area information beyond that already stored (the included cache is sufficient for the default benchmark parameters):

- Install FloPoCo 2.5.0:
  `http://flopoco.gforge.inria.fr/flopoco_installation.html`
- Install ISE Design Suite version 14.7 (requires a license file):
  `https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html`

Either add the following locations of binaries for the above programs to your `$PATH` shell environment variable, or make symbolic links and add them to an existing `$PATH` location like `/usr/bin/`:

- `/path/to/your/flopoco-2.5.0/flopoco`
- `/path/to/your/Xilinx/14.7/ISE_DS/ISE/bin/lin64/xst`

A.2 Usage

While in the project directory, run the following command to run the default parameters and see graphs:

```
PYTHONPATH=. python3 tests/fused/analysis.py
```

The function call `run()` at the end of tests/fused/analysis.py can take keyword arguments for analysis and plotting.

Appendix B

Benchmarks

For each benchmark, the expression and input ranges are given.

B.1 PolyBench

`2mm_1`

$$t + (1.5 \times a \times b)$$
$$a \in [0,1], b \in [0,1], t \in [0,3300] \tag{B.1}$$

`2mm_2`

$$d + (t \times c)$$
$$t \in [0,1], c \in [0,1], d \in [0,5940001.2] \tag{B.2}$$

`3mm`

$$e + (a \times b)$$
$$a \in [0,0.2], b \in [0,0.2], e \in [0,80] \tag{B.3}$$

`correlation`

$$s + ((d + m) \times (d + m))$$
$$s \in [0,1], d \in [0,6000], m \in [-1,0] \tag{B.4}$$

`deriche`

$$(a \times i) + (c \times x) + (b \times y) + (d \times z)$$
$$a \in [0,1], i \in [0,1], c \in [0,1], x \in [0,1], b \in [0,1], y \in [0,1], d \in [0,1], z \in [0,1] \tag{B.5}$$

`fdtd-2d_1`

$$a + (0.5 \times (c + b))$$
$$a \in [0,1], b \in [-1,0], c \in [0,1] \tag{B.6}$$

`fdtd-2d`

$$h + (-0.7) \times (e + f + y + z)$$
$$h \in [0,1], e \in [0,1], f \in [-1,0], y \in [0,1], z \in [-1,0], h \in [0,1] \tag{B.7}$$

`gemm`

$$c + (32412 \times a \times b)$$
$$a \in [0,1], b \in [0,1], c \in [0,1] \tag{B.8}$$

`gemver`

$$a + (u \times v) + (w \times x)$$
$$a \in [0,1], u \in [0,4000], v \in [0,2000], w \in [0,1000], x \in [0, \frac{2000}{3}] \tag{B.9}$$

`heat-3d`

$$'0.125 \times (a + (-2) \times b + c) + 0.125 \times (d + (-2) \times b + e) + 0.125 \times (f + (-2) \times b + g) + b',$$
$$a \in [0,30], b \in [0,30], c \in [0,30], d \in [0,30], e \in [0,30], f \in [0,30], g \in [0,30] \tag{B.10}$$

`jacobi-1d`

Note that the number 0.33333 is exactly as it appears in the PolyBench source code.

$$0.33333 \times (a + b + c)$$
$$a \in [0,1], b \in [0,1], c \in [0,1] \tag{B.11}$$

`seidel`

$$0.2 \times (a + b + c + d + e)$$
$$a \in [0,1], b \in [0,1], c \in [0,1], d \in [0,1], e \in [0,1] \tag{B.12}$$

`symm`

$$(1.2 \times c) + (1.5 \times a \times b) + (1.5 \times t)$$
$$a \in [0,5], b \in [0,5], c \in [0,5], t \in [0,1] \tag{B.13}$$

`syr2k`

$$c + (a \times 1.5 \times b) + (e \times 1.5 \times d)$$
$$a \in [0,1], b \in [0,1], c \in [0,1.56], d \in [0,1], e \in [0,1] \tag{B.14}$$

`syrk`

$$
\begin{gathered}
c+(1.5\times a*b)\\
a\in[0,1], b\in[0,1], c\in[0,1.2]
\end{gathered}
\tag{B.15}
$$

B.2 Livermore Loops

`2d_hydro`

Kernel 23—2-D implicit hydrodynamics fragment

$$
\begin{gathered}
z+(0.175\times(a\times b+c\times d+e\times f+g\times h+i+j))\\
a\in[0,1], b\in[0,1], c\in[0,1], d\in[0,1], e\in[0,1], f\in[0,1],\\
g\in[0,1], h\in[0,1], i\in[0,1], j\in[-1,0], z\in[0,1],
\end{gathered}
\tag{B.16}
$$

`state_frag`

Kernel 7—equation of state fragment

$$
\begin{gathered}
u+r\times(z+r\times y)+t\times(a+r\times(b+r\times c)+t\times(d+r\times(e+r\times f)))\\
a\in[0,1], b\in[0,1], c\in[0,1], d\in[0,1], e\in[0,1], f\in[0,1],\\
r\in[0,1], t\in[0,1], u\in[0,1], y\in[0,1], z\in[0,1]
\end{gathered}
\tag{B.17}
$$

Bibliography

[1] Y. Fukumura, P. Hamilton, M. Nakahata, and T. Oomori, "Three-term input floating-point adder-subtractor," Patent US8 185 570 B2, 2012. [Online]. Available: https://www.google.com/patents/US8185570

[2] B. Pasca, "Customizing floating-point operators for linear algebra acceleration on fpgas," Ph.D. dissertation, 2008. [Online]. Available: http://www.bogdan-pasca.org/resources/publications/Master2%20Report%20-%20Bogdan%20Pasca%202008.pdf

[3] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www.cs.ucla.edu/pouchet/software/polybench*, 2012.

[4] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Application Specific Processors, 2008. SASP 2008. Symposium on.* IEEE, 2008, pp. 101–107.

[5] R. Manohar, R. Karmazin, and C. T. O. Otero, *Automated layout for integrated circuits with nonstandard cells*, 2014.

[6] D. B. Thomas, "A general-purpose method for faithfully rounded floating-point function approximation in fpgas," in *Computer Arithmetic (ARITH), 2015 IEEE 22nd Symposium on.* IEEE, 2015, pp. 42–49.

[7] X. Gao, S. Bayliss, and G. A. Constantinides, "Soap: Structural optimization of arithmetic expressions for high-level synthesis," in *Field-Programmable Technology (FPT), 2013 International Conference on.* IEEE, 2013, pp. 112–119.

[8] X. Gao and G. A. Constantinides, "Numerical program optimization for high-level synthesis," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, 2015, pp. 210–213.

[9] X. Gao, J. Wickerson, and G. A. Constantinides, "Automatically optimizing the latency, area, and accuracy of c programs for high-level synthesis," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, 2016, pp. 234–243.

[10] W. Kahan, "Ieee standard 754 for binary floating-point arithmetic," *Lecture Notes on the Status of IEEE*, vol. 754, no. 94720-1776, p. 11, 1996.

[11] N. J. Higham, *Accuracy and stability of numerical algorithms.* SIAM, 2002.

[12] X. Inc., "Virtex-6 family overview," p. 11, 20th August 2015 2015. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[13] F. D. Dinechin and B. Pasca, "Designing custom arithmetic data paths with flopoco," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[14] J.-M. Muller, *On the definition of ulp (x)*, 2005.

[15] A. M. Smith, G. A. Constantinides, and P. Y. Cheung, "Fused-arithmetic unit generation for reconfigurable devices using common subgraph extraction," in *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on.* IEEE, 2007, pp. 105–112.

[16] E. E. Swartzlander and H. H. M. Saleh, "Fft implementation with fused floating-point operations," *Computers, IEEE Transactions on*, vol. 61, no. 2, pp. 284–288, 2012, iD: TN_ieee10.1109/TC.2010.271.

[17] J. H. Min, S.-W. Kim, and E. E. Swartzlander, "A floating-point fused fft butterfly arithmetic unit with merged multiple-constant multipliers," pp. 520–524, 2011, iD: TN_ieee10.1109/ACSSC.2011.6190055.

[18] M. Langhammer, "Floating point datapath synthesis for fpgas," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on.* IEEE, 2008, pp. 355–360.

[19] W. Kahan, "Pracniques: Further remarks on reducing truncation errors," *Commun.ACM*, vol. 8, no. 1, p. 40, jan 1965. [Online]. Available: http://doi.acm.org/10.1145/363707.363723

[20] N. J. Higham, "The accuracy of floating point summation," *SIAM Journal on Scientific Computing*, vol. 14, no. 4, pp. 783–799, 1993.

[21] U. Kulisch, "Circuitry for generating scalar products and sums of floating point numbers with maximum accuracy," Patent US4 622 650 A, 1986.

[22] A. R. Lopes and G. A. Constantinides, "A fused hybrid floating-point and fixed-point dot-product for fpgas," in *International Symposium on Applied Reconfigurable Computing.* Springer, 2010, pp. 157–168.

[23] N. Brisebarre, F. D. Dinechin, and J.-M. Muller, "Integer and floating-point constant multipliers for fpgas," in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on.* IEEE, 2008, pp. 239–244.

[24] J. Dongarra and P. Luszczek, *Livermore Loops*, ser. Encyclopedia of Parallel Computing. Springer, 2011, pp. 1041–1043.

[25] E. Jones, T. Oliphant, P. Peterson *et al.*, “Scipy: Open source scientific tools for python,” 2001–. [Online]. Available: http://www.scipy.org/

[26] F. D. Dinechin and L.-S. Didier, “Table-based division by small integer constants,” in *International Symposium on Applied Reconfigurable Computing*. Springer, 2012, pp. 53–63.