2013

Accuracy, Cost and Performance Trade-Offs for Streaming Set-Wise Floating Point Accumulation on FPGAs

Krishna Kumar Nagar
University of South Carolina

Accuracy, Cost and Performance Trade-offs for Streaming Set-wise Floating Point Accumulation on FPGAs

by

Krishna Kumar Nagar

Bachelor of Engineering
Rajiv Gandhi Proudyogiki Vishwavidyalaya 2007

Master of Engineering
University of South Carolina 2009

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosphy in

Computer Engineering

College of Engineering and Computing

University of South Carolina

2013

Accepted by:

Jason D. Bakos, Major Professor

Duncan Buell, Committee Member

Manton M. Matthews, Committee Member

Max Alekseyev, Committee Member

Peter G. Binev, External Examiner

Lacy Ford, Vice Provost and Dean of Graduate Studies

Dedication

Dedicated to

Dearest Maa, Paa and beloved Prakriti

Acknowledgments

It took me 6 years, 2 months and 27 days to finish this dissertation. It involved the support, patience and guidance of my friends, family and professors. Any words of gratitude would not be sufficient for expressing my feelings.

I would like to thank my committee members Dr. Duncan Buell, Dr. Manton Matthews, Dr. Max Alekseyev and Dr. Peter Binev. Their guidance, suggestions and feedback at various instances has greatly benefited this work and my knowledge. I would also thank them for patiently reading this document and be approving of this research.

It would have been impossible for me to pursue the research without the guidance and instruction of Dr. Jason Bakos. He not only has been supportive but also encouraging. His dedication, commitment and enthusiasm kept me motivated and inspired. Foremost, he has been patient with me throughout. I cannot imagine a better mentor, guide and adviser for my PhD.

I would like to express my gratitude towards my colleges in the Heterogeneous and Reconfigurable Computing Lab especially Zheming Jin and Yan Zhang with whom I worked on various projects. I would also thank Ibrahim Savran, with whom I prepared for the qualifying exam and had discussion on various many research topics.

My family has always provided me unconditional support and have always kept me motivated. I sincerely appreciate their support.

Abstract

The set-wise summation operation is perhaps one of the most fundamental and widely used operations in scientific applications. In these applications, maintaining the accuracy of the summation is also important as floating point operations have inherent errors associated with them. Designing floating-point accumulators presents a unique set of challenges: double-precision addition is usually deeply pipelined and without special micro-architectural or data scheduling techniques, the data hazard that exists. There have been several efforts to design floating point accumulators and accurate summation architecture using different algorithms on FPGAs but these problems have been dealt with separately. In this dissertation, we present a general purpose reduction circuit architecture which addresses the issues of data hazard and accuracy in set-wise floating point summation. The reduction circuit architecture is parametrizable and can be scaled according to the depth of the adder pipeline. Also, the dynamic scheduling logic we use in this makes it highly resource efficient. Further, the resource requirements for this design are low. We also study various methods to improve the accuracy of summation of floating point numbers. We have implemented four designs. The reduction circuit architecture serves as the framework for these designs. Two of the designs namely AEC and AECSA are based on compensated summation while the two designs called EPRC80 and EPRC128 implement set-wise floating point accumulation in extended precision. We present and compare the accuracy and cost- operating frequency and resource requirements- tradeoffs associated with these designs. On the basis of our experiments, we find that these designs achieve significantly better accuracy. Three of the designs– AEC, EPRC80

and EPRC128– operate at around 180MHz on Xilinx Virtex 5 FPGA which is comparable to the reduction circuit while AECSA operates at 28% less frequency. The increase in resource requirement ranges from 41% to 320%. We conclude that accuracy can be achieved at the expense of more resources but the operating frequency can be maintained.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The floating point summation operation– or accumulation–, $S = \sum_{i=1}^{n} a_i$ is one of the most widely used operations in scientific applications such as computational fluid dynamics, structural modeling, power networks and it also forms the core of most of the basic linear algebra subroutines(BLAS)[39]. In a simplistic scenario, accumulation can be performed using the following snippet of code:

```
for(i = 1; i < n+1; i++) {
        S = S + a[i];
}
```

In many applications, like sparse matrix vector multiplication, it is required to accumulate streaming datasets of different sizes. The applications also require high numerical accuracy. Achieving high performance for the accumulation operation presents two challenges. Firstly, on a parallel computer, the summation can be performed using a parallel "reduction" operation. In this operation, multiple additions are performed concurrently, producing multiple "partial" sums that are maintained separately until they are eventually added into a final result. The problem with this approach is that since floating point addition is not associative and different orders in which the additions are performed lead to different rounding errors. This means that the error encountered during the reduction is associated with its runtime behavior.

The second challenge arises from the fact that floating point adders are deeply pipelined. If a new input is delivered to the adder every clock cycle, it cannot provide the sum of the current inputs before the next value arrives, thus creating a data

hazard. In such a case, the designer cannot use a simple feedback-based accumulator as it may result in addition of values belonging to different sets. In order to accumulate multiple sets of different sizes and deal with data hazard, a special circuit with proper scheduling techniques around floating point adders is required. This requirement leads to control overhead which may require additional resources and may reduce the hardware utilization.

Implementing parallel reduction operations on FPGAs provides some unique opportunities to maintain and improve the accuracy and resolve the issue of data hazard. A rich set of tradeoffs can be made between error bound and resource usage through the design of customized floating point units. Also, very fine-grained control logic can be used to reduce the overheads of data dependent behavior.

Several methods have been developed and studied in order to improve the accuracy of summation. Some of these methods rely on the ordering of the values in the dataset. These methods require a priori knowledge of the dataset according to which the dataset can be sorted. Methods have also been developed in which the error during addition is extracted using floating point operations. This error is incorporated in the summation results. Another way to improve accuracy is to perform all the intermediate calculations in extended precision which mitigate the effect of shift and round operations by proving more guard bits during addition. Some of the accuracy improving techniques have also been implemented on FPGA based coprocessors. Also, there have been several efforts to implement streaming set-wise reduction on FPGAs.

All the methods to improve accuracy come at the cost of increased number of operations. Techniques which require reordering the inputs are not practical for implementation on FPGAs as the upfront cost of reordering is very high which affects the overall throughput adversely. Also, these methods are not suitable where streaming datasets are used as reordering the dataset is not possible. Other methods in which the errors are calculated and incorporated in the result(s) require additional floating

point operations to extract the error and hence the latency can be large. Performing intermediate calculations in extended precision requires wider and deeper adder unit. Also, conversion units are required to convert the input to extended precision and output to native precision format. Methods which rely on error extraction and compensation as well as those in which extended precision is used are comparatively less expensive than reordering as dataset need not be pre-processed. Also, a priori knowledge of the dataset is not required.

The motivation of this research arises from the observation that previous works address the problem of improving accuracy and streaming set-wise reduction of floating point values separately. In this dissertation, we present a high performance general purpose reduction circuit which addresses the issue of data hazard and scheduling associated with set-wise floating point accumulation. This reduction circuit achieves nearly 100% utilization regardless of the structure of the input dataset. We also examine methods to leverage custom hardware in order to place tighter error bounds on the accumulation operation. On the basis of our analysis, we have implemented four designs for accurate summation of set-wise data. Our reduction circuit architecture serves as the basic framework for these designs. Two of the designs are based on compensated summation methods while in two designs we perform all the intermediate operations in extended precision.

In compensated summation methods, additional floating point operations are required which essentially increase the overall latency of the adder network. Due to the increased latency, additional resources are required to store the partial sums and the control logic becomes complex. In order to reduce the number of floating point operations, we have designed a custom floating point adder which not only adds two input values but also produces the rounding error which occurs during the floating point addition. The designs in which we emulate compensated summation methods, make use this custom adder to obtain the error required for compensation.

We analyze these designs and evaluate these in terms of accuracy, cost and performance. Though, we require more resources for all these designs, but we observe less impact on the overall performance in terms of operating frequency. We also report and compare the accuracy of results from simulation of these designs with simulation of original reduction circuit and software model of extended precision recursive summation. We conclude that new designs achieve better accuracy than the original reduction circuits and simple recursive summation method.

The rest of the document is organized as follows: The Chapter 2, we introduce FPGAs, IEEE-754 floating point standard and floating point addition. In Chapter 3, various methods to improve accuracy of floating point summation and implementations on FPGAs and implementations of set-wise accumulation of floating point values have been studied. Chapter 4 describes the design our reduction circuit. In Chapter 5, we present the implementation details of the new designs for improving the accuracy of set-wise floating point summation. In Chapter 6, we discuss the accuracy parameters based on our experiments and the tradeoffs associated with the designs.In Chapter 7, we summarize the dissertation, and discuss the conclusion and future work.

Chapter 2

Background

2.1 Field Programmable Gate Array

Field programmable gate arrays (FPGAs) are configurable logic devices which consist of arrays of configurable logic blocks (CLBs) connected through programmable interconnects. Modern FPGAs also consist of hardwired logic blocks such as RAMs and DSPs. Application specific logical functions can be implemented and updated on these devices.

In reconfigurable computing, kernel computations are performed using special purpose architectures that are implemented on one or more FPGAs. Designs on FPGAs generally achieve better performance than software implementations on general purpose processors because as every hardware component is dedicated to the computation. It is similar to designing application specific integrated circuits (ASIC) except that the resultant frequency on FPGA(s) is about 10x slower than ASICs.

2.2 Floating Point Representation- IEEE-754 Standard

IEEE-754 floating point standard has been developed in order to simplify the arithmetic algorithms designed for floating points[16]. The standard has been widely adopted and thus it ensures interoperability and portability of algorithms on different machines. The representations are characterized by base(β), precision(p) and exponent range. It allows representation of floating point numbers using base 2 (binary) and base 10 (decimal). IEEE-754 single and double precision binary formats

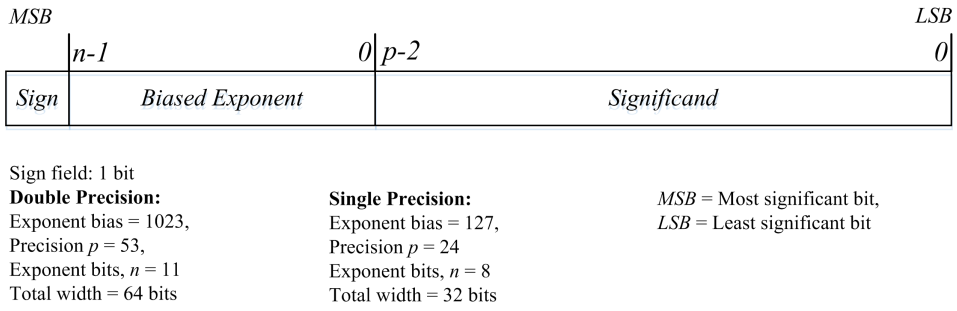


Figure 2.1: Binary Interchange Format

are widely used while the support other formats is still limited.

In IEEE-754 binary formats, apart from special cases, the digit to the left of the binary point is 1 which is often referred as leading one. This notation is called normalized form. Figure 2.1 depicts the encoding for a floating point number in the binary standard. It is called the binary interchange format in which each number has only one encoding. It has three fields- sign, exponent and significand. The leading one is not included but is implied in calculations. Also, in order to simplify operations for exponents like comparison, a bias is used such that the exponent can be treated as an unsigned integer.

A floating point number A encoded in IEEE-754 format can be calculated as

$$\hat{A} = (-1)^{Sign} \times (1 + Significand) \times 2^{(Exponent - Bias)}. \tag{2.1}$$

where $\hat{A}$ is the floating point representation of A. Since the number of digits or bits used for representing a fraction is fixed and limited, all numbers cannot be represented exactly in these formats. Thus the encoding of a number in floating point format its approximation to the nearest floating point number. That is why floating point format is often termed as limited precision floating point format as well. For example,

$$0.062371 = 6.2371 \times 10^{-2} \text{ with 5 digit precision,}$$

$$0.062371 = 6.24 \times 10^{-2} \text{ with 3 digit precision.}$$

Apart from these standard formats, in order to increase the precision, IEEE-754 also allows extended precision format in which the precision and range of exponent is defined by the user.

2.3 Floating Point Addition

A number of steps are involved in addition of two floating point values[21, 36]. The implied one, not included in the representation, is also taken into account. Flowchart in figure 2.2 depicts the algorithm for addition of two floating point numbers.

First, the exponents of the values are compared. If the exponent of the first input is smaller then the operands are swapped so that the following shift operation is restricted in one path of the hardware unit. Then the significand of number with smaller exponent aligned is with the other significand and the significands are added. The result is then normalized by shifting it either to left or right and the shift amount is added or subtracted from the exponent. The resultant is rounded so that it can be put back in the standard format. Rounding is used to represent the result in the nearest possible floating point format. Multiple iterations may be required to round the result properly as the rounded result may require normalization. When implementing this floating point addition in hardware, extra bits can be used in order to improve the accuracy. IEEE-754 standard supports various rounding modes such as round-ties-to-even, round-ties-to-away, round-toward-positive, round-toward-negative and round-toward-zero. In this dissertation, we only consider round-ties-to-even as the methods we use work only with round-to-nearest mode. The reasons have been explained later.

During floating point addition, extra bits namely guard ($\mathbb{G}$), round ($\mathbb{R}$) and sticky ($\mathbb{S}$) are used to perform addition of mantissas. Guard bit is the 2nd, round bit is the 1st and sticky bit is the 0th bit in the extended mantissa. In round-to-nearest mode, we consider values of 3rd least significant bit (including the extra bits) of mantissa, $\mathbb{L}$

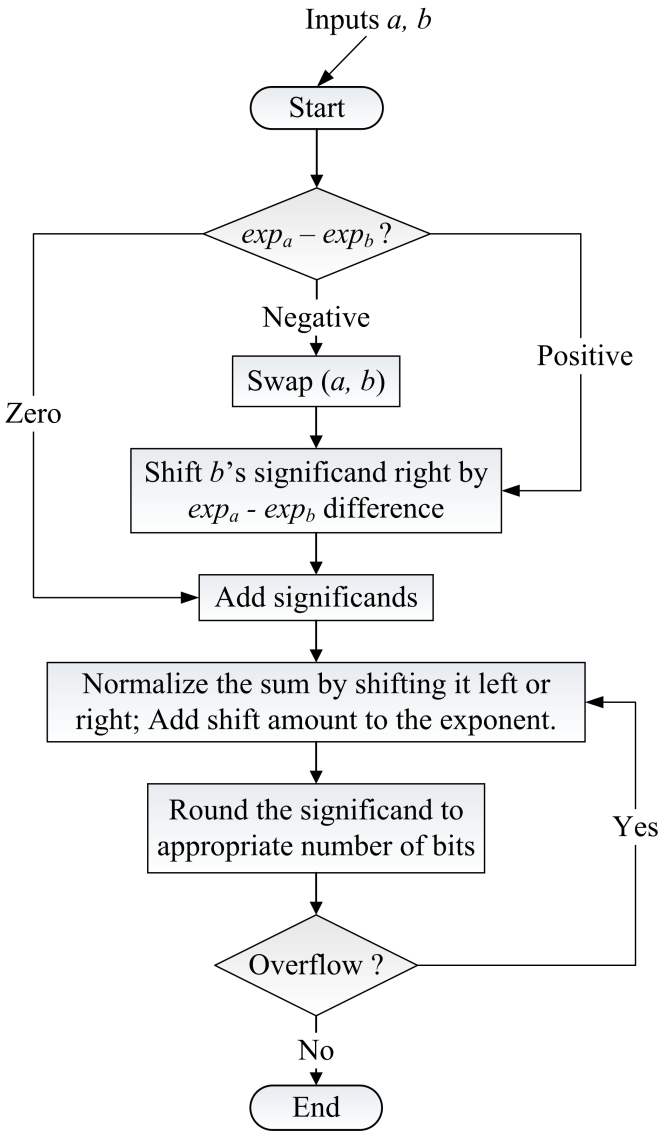


Figure 2.2: Floating Point Addition Algorithm

and $\mathbb{G}$, $\mathbb{R}$ and $\mathbb{S}$ bits. The direction of rounding- round-up or round-down- is decided as depicted in Table 2.1.

It can be observed that when $\mathbb{G}$ is 0, sum is rounded down i.e. towards zero. If $\mathbb{G}$ is 1, then $\mathbb{R}$ and $\mathbb{S}$ are also considered in the rounding decision. $\mathbb{L}$ is only considered in case of a tie i.e. when $\mathbb{G}$=1, $\mathbb{R}$=0 and $\mathbb{S}$=0, $\mathbb{L}$, otherwise it is in "*don't care*" (X) state.

Since floating point addition involves a number of atomic operations, it is generally

Table 2.1: Round To Nearest

$\mathbb{L}$	$\mathbb{G}$	$\mathbb{R}$	$\mathbb{S}$	Round
X	0	X	X	Roun Down
X	1	0	1	Round Up
X	1	1	0	Round Up
X	1	1	1	Round Up
0	1	0	0	Round Down (Tie)
1	1	0	0	Round Up (Tie)

implemented as deeply pipelined hardware unit where each step may take one of more clock cycle for completion. Thus, the latency of a floating point adder is more than one cycle.

2.3.1 (In)Accurate Floating Point Addition

Methods, in which large floating point datasets are used, may deliver wrong results due to different sources of errors. Stability of numerical algorithms is an important topic of research in the field of numerical analysis and much of the focus has been given to the accuracy of floating point operations [1, 2, 3].

Rounding errors, also called round-off errors, are unavoidable due to prevalence of finite precision floating point arithmetic[12]. Rounding error can be introduced in two ways- shift and carry. The error due to shift operation of smaller operand during addition causes shifting error. A nonzero carry which results the significand width to be more than the allowed bits causes carrying error.

Real numbers are represented as the nearest possible approximation to a floating point representation but floating point operations are performed as if the intermediate results are correct to infinite precision and then rounded for the given format. Thus the error in a floating point operation involving two numbers can be calculated using the following equation.

$$fl(a \bigodot b) = (1 + \varepsilon)(a \cdot b), \quad |\varepsilon| \leqslant \epsilon, \quad \epsilon = \frac{1}{2}\beta^{1-t} \tag{2.2}$$

where fl(a $\odot$ b) is a floating point operation $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$ equivalent to the exact operation $\cdot \in \{+, -, \times, /\}$ and β is the base. This is the standard model for numerical error analysis attributed to Wilkinson[19]. ϵ is called machine precision or unit roundoff. It gives an upper bound for relative error due to rounding in floating point arithmetic. It is 2^{-24} for single precision and 2^{-53} for double precision.

Equation 2.2 is true only for rounding to nearest rounding mode. In this rounding mode,

2.3.2 Rounding Error in Summation

As the number of additions increase, the total error in the result also increases. As shown in figure 2.3, floating point addition is not associative hence each of the possible orderings may produce a different summation and the corresponding error may vary greatly as well. Conversely, the summation calculated in one way may be more accurate than the summation calculated in another way. The error bound for recursive summation is given by the following equation[2]:

$$|E_n| \leqslant (n-1)\epsilon \sum_{i=1}^{n} |a_i| + O(\epsilon^2) \tag{2.3}$$

This error bound has been deduced by taking into consideration the error introduced in each iteration step of the recursive addition and thus is dependent on the number of terms being added. It is independent of the order in which the values are added. In order to reduce the error in the result, the input values can be reordered such that the final summation is more accurate and the error bound given by the equation is minimized.

Given a set of n numbers, the number of ways in which they can be added is given by:

$$C_n = \frac{1}{(n+1)}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \tag{2.4}$$

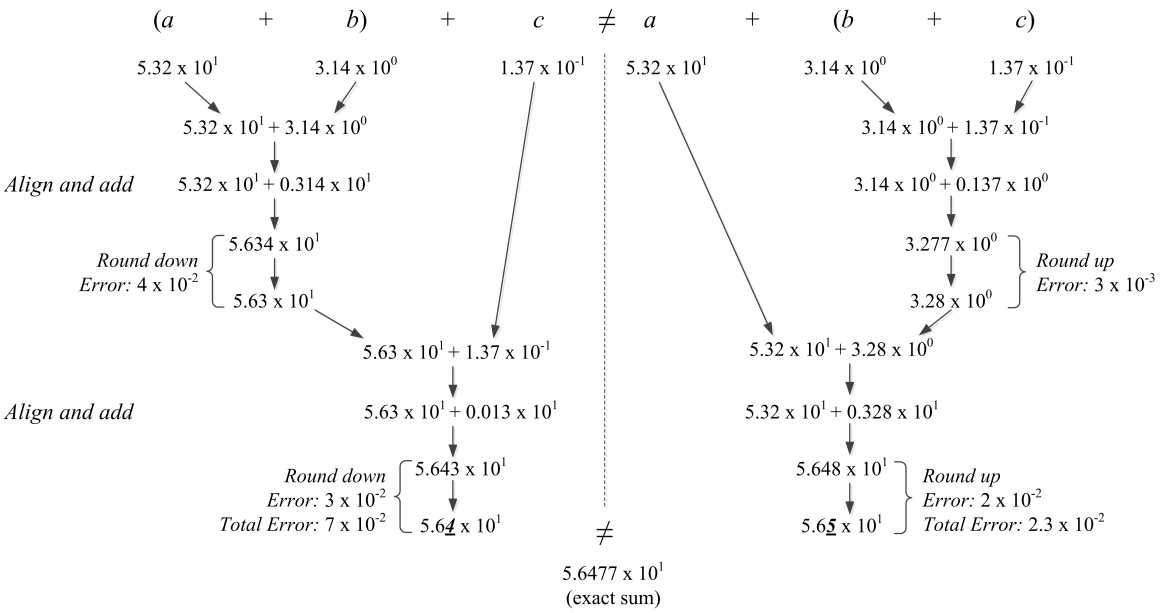


Figure 2.3: Non-associativity of Floating Point Addition

where C_n is called the Catalan number. Since this number grows very large even with small number of values, it is not possible to try every possible ordering and choose the one which results in the least error.

Chapter 3

Previous Work

In this chapter, we summarize various methods to improve accuracy of floating point summation, implementations of these methods on FPGAs and implementation of set-wise floating point summation on FPGAs.

3.1 Methods to Improve Accuracy of Floating Point Summation

In this section, we discuss different methods to improve accuracy. First, we discuss the dependence of error bounds and relative error on the distribution of value in a dataset. Then, we explain the effects of reordering the data on the summation error. Then we discuss the methods in which the error encountered during floating point addition is calculated and incorporated into the final summation. We also present a method in which extended precision is used to improve the accuracy of the intermediate operations.

3.1.1 Relative Error and Condition Number

In numerical error analysis, the error bounds are derived as if the worst case error occurs always and is propagated throughout the computation. For this to happen, rounding during all intermediate operations should be in same direction. In general, the rounding direction is random and the errors often cancel out each other. Thus the magnitude of final error can be very small. Also, the magnitude of error also depends on the order of values being added. Hence, the relative error with respect to summation result is taken into consideration instead of the absolute value of the

error. The relative error is the ratio of the error,$|E_n|$ and the summation calculated with infinite precision as depicted in equation 6.2. In this equation, the error, $|E_n|$ is the difference between the exact summation calculated with infinite precision, $|S_n|$ and the recursive summation in floating point precision, S_n.

$$\frac{|E_n|}{|S_n|} = \frac{|S_n| - S_n}{|S_n|} \tag{3.1}$$

The ratio of summation of absolute values and the recursive summation denotes the condition number, κ, of the dataset and is calculated as in equation 6.1.

$$\kappa = \frac{\sum_{i=1}^{n} |a_i|}{|\sum_{i=1}^{n} a_i|} \tag{3.2}$$

If a dataset consists of only positive values then $\kappa = 1.0$ while $\kappa > 1.0$ denotes that the dataset consists of both positive and negative values. The relative error bound of summation methods is proportional to κ. If $\kappa \gg 1.0$, the relative error can be large even for methods used for achieving high accuracy. Thus, a low error bound does not always lead to low relative error.

3.1.2 Reordering Inputs to Achieve Better Accuracy

In floating point summation, the error in final result has been attributed to several factors. Catastrophic cancellation occurs when the final result is much smaller than the intermediate sums. It is due to the presence of large intermediate sums with opposite signs. A method has been proposed in which the numbers are added to numbers with same sign and the final sum is calculated by adding the result from each group of numbers. Though this method eliminates the intermediate cancellations, it results in a cancellation at the end and hence is not effective in error correction.

Large relative errors can also occur when the intermediate results become much larger than the individual operands. This is caused by the presence of several intervals in the data-sets. In order to reduce such errors, the numbers can be arranged

Table 3.1: Mean Square Errors for Different Orderings

Distribution	Increasing	Random	Decreasing	Insertion	Pairwise
Uniform(0, 2μ)	$0.20\mu^2n^3\sigma^2$	$0.33\mu^2n^3\sigma^2$	$0.53\mu^2n^3\sigma^2$	$2.6\mu^2n^2\sigma^2$	$2.7\mu^2n^2\sigma^2$
Exp(μ)	$0.13\mu^2n^3\sigma^2$	$0.33\mu^2n^3\sigma^2$	$0.63\mu^2n^3\sigma^2$	$2.6\mu^2n^2\sigma^2$	$4.0\mu^2n^2\sigma^2$

Note: This is based on the assumption that relative errors in floating point addition are statistically independent and have zero mean and finite variance σ^2. Uniform distribution, Unif(0, 2μ), and exponential distribution, Exp(0,μ) have mean μ.

according to their intervals. Operations can be performed within those intervals and final result can be calculated by adding results from those intervals[14]. For distributing the values in intervals and adding the values from different intervals, the values and summation of intervals can be reordered in different ways to reduce the overall error.

Several possible orderings have been studied for improving the accuracy of the summation. It has been observed that if the numbers are added in absolute ascending order, the error bound is minimized. This results in addition of operands having similar magnitude and hence reduces the carrying error[20]. But if the dataset contains both positive and negative numbers, this method results in a greater error. Absolute descending order is more beneficial when there is lot of cancellation i.e. positive and negative numbers are evenly present in the dataset as it reduces the carrying error.

Another method applicable to positive numbers called the insertion method has been proposed in which the intermediate partial sum is inserted in the sorted list of numbers. This is effective in reducing the shifting error but does not work well for dataset having both positive and negative numbers[6]. In pairwise summation, operands are added in pairs. This method is suited for parallel implementation with binary tree of adders. The final result can be calculated in $log_2 n$ steps as opposed to *n-1* steps required for serial recursive summation at the expense of resources[3, 5].

Table 3.1 lists the mean square errors over uniform and exponential distributions for different methods as studied in [13].

In the above mentioned methods, certain orderings were considered so as to reduce the amount of error in the final result. These methods do not calculate or take into consideration the error which occurs. The input set size and magnitudes have to be known a priori. Also, these methods require sorting and arrangement of data which are expensive operations and require more resources. Due to the upfront cost of sorting the input data, the throughput of the summation is affected adversely. Further, the error bounds of these methods are data dependent. Thus these methods are not suitable for high performance implementations.

3.1.3 Compensated Summation

Several methods have been developed where the error during each floating point addition can be calculated and incorporated into intermediate and the final results[15]. Such methods are categorized as compensated summations. Compensated summations are based on error free transformation which is a transformation in which

$$a + b = x + e, \qquad x = fl(a \bigoplus b). \tag{3.3}$$

Here x is the rounded floating point sum of two floating point numbers a and b represented in base 2 and e is the error incurred during addition. The error e can be recovered using floating point operations. During addition, the operand with smaller exponent, b is shifted by the exponent difference for alignment. The low order bits of b, b_l are discarded. These bits form the error term. Figure 3.1 shows the general idea behind the recovering b_l [1]. Floating point operation, x - a results in high order bits of b, b_h. Subtracting b from b_h results in the low order bits of b_l. The recovered low order bits can be incorporated in subsequent additions to get more accurate result.

Algorithm 1 lists Kahan's compensated summation algorithm which calculates and applies the correction in each iteration for recursive summation[17]. In this algorithm, the error term e - the approximation to the rounding error, is subtracted from the next input value in subsequent iteration. Another step can be added to this

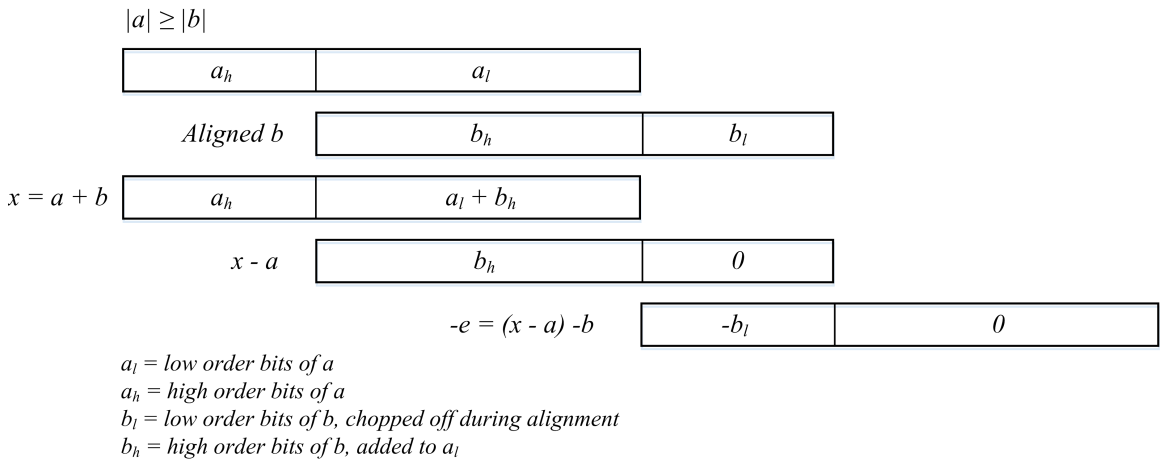


Figure 3.1: Error Free Transformation- Error Recovery

algorithm in which $S + e$ can be calculated at the end of the loop [18]. It has been shown that this algorithm improves the error bound and gives almost ideal result. Equation 3.4 gives the error bound for Kahan's compensation algorithm which is independent of n if $n\ \epsilon < 1$.

$$|E_n| \leqslant (2\epsilon + O(n\epsilon^2)) \sum_{i=1}^{n} |a_i| \tag{3.4}$$

Equation 3.5 gives the relative error bound for Kahan's compensation algorithm.

$$\frac{|E_n|}{|S_n|} \leqslant (2\epsilon + O(n\epsilon^2)) \frac{\sum_{i=1}^{n} |a_i|}{|\sum_{i=1}^{n} a_i|} \tag{3.5}$$

Another version of compensated summation has been developed where the error terms calculated after each addition are accumulated and the correction is applied at the end of the summation. The error at each step of recursive addition can be calculated using Fast2Sum[11] algorithm as depicted in algorithm 2. Algorithm 3 can be used to calculate the final sum. Equation 3.6 shows the error bound for this algorithm.

$$|E_n| \leqslant (2\epsilon + O(n^2\epsilon^2)) \sum_{i=1}^{n} |a_i| \tag{3.6}$$

The relative error for this method is given by equation 3.7.

$$\frac{|E_n|}{|S_n|} \leqslant (2\epsilon + O(n^2\epsilon^2)) \frac{\sum_{i=1}^{n} |a_i|}{|\sum_{i=1}^{n} a_i|} \tag{3.7}$$

Algorithm 1 Kahan's Compensated Summation Algorithm

1: $input(a_1, a_2, ..., a_n)$
2: $S = a_1$
3: $e = 0.0$
4: **for** $i = 2 : n$ **do**
5: **if** $|S| < |a_i|$ **then**
6: $swap(a_i, S)$
7: **end if**
8: $y = a_i - e$
9: $S_t = S + y$
10: $e_t = S_t - S$
11: $e = e_t - y$
12: $S = S_t$
13: **end for**
14: $Return(S)$

Algorithm 2 Fast2Sum Algorithm

1: $Input(a, b)$
2: **if** $|a| < |b|$ **then**
3: $swap(a, b)$
4: **end if**
5: $x = a + b$
6: $b_t = x - a$
7: $e = b - b_t$
8: $Return(x, e)$

Algorithm 3 Error Compensation with Fast2Sum

1: $Input(a_1, a_2, ..., a_n)$
2: $S = a_1$
3: $e = 0.0$
4: **for** $i = 2 : n$ **do**
5: $(S, e_i) = Fast2Sum(S, a_i)$
6: $e = e + e_i$
7: **end for**
8: $S = S + e$
9: $Return(S)$

Kahan's compensation method can also be expressed using Fast2Sum as depicted in algorithm 4.

It must be noted that Kahan's compensation algorithm and Fast2Sum algorithm

Algorithm 4 Kahan's Compensated Summation with Fast2Sum

1: $Input(a_1, a_2, ..., a_n)$
2: $S = a_1$
3: $e = 0.0$
4: **for** $i = 2 : n$ **do**
5: $\quad y = a_i + e$
6: $\quad (S, e) = Fast2Sum(S, a_i)$
7: **end for**
8: $Return(S)$

require that $|a| \geqslant |b|$. This essentially creates a branch in software implementations of these algorithms but when implementing these in hardware, the swap operation in floating point adder eliminates the need of this check and thus accurate summation can be calculated with three additional steps.

Several other variations compensated summation techniques have been developed but require significantly more number of floating point operations [4, 7, 8, 9, 10, 35].

It can be observed that in compensated summation methods, additional steps are required to recover the error encountered during the alignment operation. But these do not require a priori knowledge of the data hence they are more suitable and less expensive for hardware implementation.

3.1.4 Intermediate Operations in Extended Precision

Performing the intermediate calculations in higher precision can also be useful in achieving higher accuracy [22, 23]. For example, on x86 CPUs, the floating point numbers denoted in single and double precision are converted to 80-bit extended precision format and the operations are performed in this format. Computations in extended precision format lead to results at least as accurate as 64-bit results. In other words, the error in result from extended precision cannot be more than that from double precision. This is because extended precision format ensures presence of more guard bits and hence improves the rounding error. The numbers are converted back to respective format before storing them. Use of extended precision format can

lead to more accurate results without much impact on the performance.

All the aforementioned methods to improve the accuracy of summation operations require additional additional operations. Methods which require reordering the dataset need sorting which is a complex task. Compensated summation methods require additional floating point operations for calculating and incorporating the error. Wider adder and intermediate storage units are required during calculations in extended precision. Thus all the methods to improve accuracy come at additional cost of resources and floating point operations.

3.1.5 Implementation on FPGAs

There have been several attempts to implement the methods which improve the accuracy of floating point summation on FPGAs. When designing accurate accumulators, approaches can be broadly divided in two categories- those that reorder the inputs to reduce the errors and those that use error free transformations to find and correct the errors.

In the reordering approach the properties of the set to be accumulated-sign and magnitude of values and size of the input set- are known a priori in order to preprocess the data. A group from Texas A&M University designed a custom adder for accumulation[32]. In this implementation, exponents of all the inputs are compared and the significands are aligned for the greatest exponent. Then the significands are added as fixed point numbers. Though this method reduces the error bound, it comes with the overhead of sorting and storing all the inputs. Also, wide integer addition is a slow operation, hence width of the aligned significands becomes a limiting factor.

Another approach has been described in [31] to get the same results as in-order inputs. The sum is calculated using parallel prefix reduction network and verified after each reduction step. If the addition is not same as in-order sum, the calculated error is propagated in the reduction network in a recursive manner until equal result

is achieved. While this leads to the final summation equal to that in original order but multiple cycles may be required to converge to zero error. Further, static scheduling is required along with a fixed number of inputs in a set. Also, this does not reduce the error in the summation operation.

In order to achieve better accuracy in floating point summation, an iterative distillation algorithm involving Fast2Sum for error free transformation was implemented on FPGAs in [33]. The number of iterations required to achieve the accuracy depends on the number of values and nature of the data in the set. A custom adder was also developed to reduce the number of floating point operations and dependence on data set but the resource utilization was reported to be 47% and 121% higher than Fast2Sum implementation for single and double precision custom adders respectively. Also, the performance remains the same for single precision and degrades for double precision. Thus, this approach does not scale well with precision.

The most recent approach to implement accurate summation operation has been described in [34]. Here, a binary tree network is used for summation. Custom adders have been used in the binary tree which output the residue error resulting from the shift operation in addition along with the sum of inputs. These errors are accumulated using another binary tree and are added to the result emerging from the main binary tree iteratively. The number of iterations required for convergence of error are not bound and may vary significantly with data. Also, two binary tree networks are used and hence the number of resources required are very high.

3.2 Set-wise Floating Point Summation on FPGAs

In large system solvers such as Conjugate Gradient, in which sparse input data is used, the data is divided into multiple sets and these sets are generally of different sizes. The input values generally have a DII of one cycle and and sets are not intermixed. We refer to them as streaming datasets. The implementations for improving the accuracy

of summation do not address the issue of data hazard and cannot handle streaming datasets. In order to accumulate different data sets represented in floating point values, a special architecture is required which not only is able to handle multiple sets simultaneously but also addresses the issue of data hazard which occurs due to deeply pipelined floating point adders. In this section, we discuss such architectures.

There have been several attempts in designing FPGA-based double precision accumulators for streaming data. These approaches can be broadly divided in two categories- static scheduling and dynamic scheduling.

3.2.1 Static Scheduling Approach

A notable example of static scheduling approach was presented by deLorimier et al[30]. Here the input values and partial sum belonging to different sets are interleaved such that consecutive values belonging to each set are delivered to the accumulator at a period corresponding to the pipeline latency of the adder. The accumulator in this case can be designed as a simple feedback adder. This allows the adder to accept a new value every clock cycle while avoiding the accumulation data hazard among values in the same accumulation set. Unfortunately, this method requires a large up-front cost in scheduling input data and is not practical for large data sets. The requirement of computation and communication scheduling technique makes the architecture's performance highly dependent on the structure of the input data. Also the memory requirement for such designs is high in order to save the partial sums.

3.2.2 Dynamic Scheduling Approach

The second approach is to use a dynamic scheduling technique that selects the input to the adder in runtime. This requires managing the progress of each active accumulation set using a controller which manages the input to the adder. Dynamic scheduling approach can be divided in two categories- those which use standard adder and those

in which the reduction circuit is integrated within custom floating point adder. In the following sections, we discuss designs based on these two approaches.

3.2.2.1 Reduction Circuit Around Standard Adder

Prasanna's group at the University of Southern California as written several seminal papers in this area [24, 25]. Their first design was a collapsed a binary adder tree organized as a linear series of adders where the number of adders scaled up as a logarithmic function of the maximum number of expected input values to be accumulated. Each adder in the system saw an exponentially lower utilization than the adder before it. This design also had a long latency, had to be flushed between input sets, and the maximum input set size was fixed at design time. As a result, this design was resource inefficient and had very low adder utilization.

Their follow-up design was based on the notion of using a single adder to coalesce an accumulation set while another adder begins reading the next input set. This design required only two adders, but its FIFO would overflow when the size of the input sets were, on average, less than $\alpha \lceil log_2 \alpha + 1 \rceil$ values each, where α is the adder latency. Also, the controller complexity required by this reduced its maximum speed by nearly 20%, relative to the maximum speed of the floating-point adder that it was built around.

Their final design overcame this limitation and required only one adder but also required two memories of size α^2 and a control overhead speed reduction of about 3%. Both of these designs had extremely complex controller overhead which limited their operating speed and effective throughput.

An implementation from UT-Knoxville and Oak Ridge National Laboratory used the collapsed binary tree approach but with a parallel–as opposed to a linear–array of α adders. This implementation striped each consecutive input across each adder in turn, achieving a fixed utilization of $1/\alpha$ for each adder. This design consumes

significantly more resources as it requires multiple adders. Also, the resources are poorly utilized.

Huang et al[26] present three reduction circuit designs based which also use the collapsed binary tree approach. The first design- modular fully pipelined architecture has $\lceil log_2\alpha \rceil + 1$ adders connected in a chain. There is a buffer associated with each of the adders where the incoming value (first adder) or partial sum from the previous adder is stored and added to the next value. The last adder has a feedback loop associated with it. This approach is very similar to a collapsed binary tree of adders and provides pairwise addition of input values. The utilization ratio of each adder is reduced at each stage of the chain. Also, the accumulator is allowed to reduce only one set at a time hence the following sets have to wait for complete reduction of one set. The second design uses two adders- one as partial sum generator while the other as accumulator. But it has a chain of FIFOs connected with the first adder which are used to delay the incoming values so in order to match the latency of the floating point adder. Role of adder in the accumulator remains the same. The third design is similar to the AeMFPA but changes the structure of FIFOs used. These three designs do not allow multiple sets to enter the accumulator stage thus require chain of adders or FIFOs to compensate for the latency of accumulate operation. Further the utilization of the adders is not optimal as they remain ideal for several cycles during the reduction process. Also, the resource utilization is very high due to the requirement of multiple adders and FIFOs. Thus the overall throughput of the reduction circuit suffers.

An improved single-adder streaming reduction architecture was later developed at the University of Twente[28, 29]. This design requires less memory and less complex control than the designs discussed previously. In this design, the adder is connected to two deep input and output buffers. The inputs to the adder are governed by a set of rules in order of priority.

Table 3.2 summarizes the some of the above discussed architectures in terms of resources- adders, buffers and latency. Note that we do not compare resource usage and performance because these factor vary with the change in platform used for implementation.

Table 3.2: Comparison of Different Reduction Circuits

Reduction Circuit	# Adders	Buffer Size	Latency
FCBT[24, 25]	2	$3\lceil log_2 \text{n} \rceil$	$2\text{n} + (\alpha\text{-}1)\lceil log_2 \text{n} \rceil$
DSA[24, 25]	2	$\alpha\lceil log_2 \alpha + 1 \rceil$	$\alpha\lceil log_2 \alpha + 1 \rceil$
SSA[24, 25]	1	$2\alpha^2$	$2\alpha^2$
Gerards[28, 29]	1	$3\alpha + \alpha\lceil log_2 \alpha \rceil + 2$	$2\alpha + \alpha\lceil log_2 \alpha \rceil + 1$
MFPA[26]	α	$\lceil log_2 \alpha \rceil + 1$	$< \text{n} + \alpha\lceil log_2 \alpha + 2 \rceil$
AeMFPA[26]	2	$\alpha\ log_2 \alpha + 2\alpha$	$< \text{n} + \alpha\lceil log_2 \alpha + 2 \rceil$

3.2.2.2 Integrated Reduction with Custom Floating Point Adder

In each of the above discussed work, standard adders (usually generated with Xilinx Core Generator) have been used as the core of the accumulator. Another approach is to design a custom adder such that the de-normalization and significand addition steps have a single cycle latency, which makes it possible to use a feedback without scheduling. To minimize the latency of denormalize portion, which includes an exponent comparison and a shift of one of the significands, both inputs are base-converted to reduce the width of exponent while increasing the width of the mantissa[27]. This reduces the latency of the denormalize while increasing the adder width. Since wide adders can be achieved cheaply with carry-chained DSP48 components, these steps can sometimes be performed in one cycle. This technique is best suited for single precision operands but can be extended to double precision as well as shown in [37].

Chapter 4

Preliminary Work

In this section we present an approach to overcome the issue of data hazard in set-wise accumulation of floating point values. Apart from this primary goal, we also focus on simplifying the control logic and reducing the amount of memory and resources required in order to achieve high performance. While designing the reduction circuit, we consider the following set of constraints:

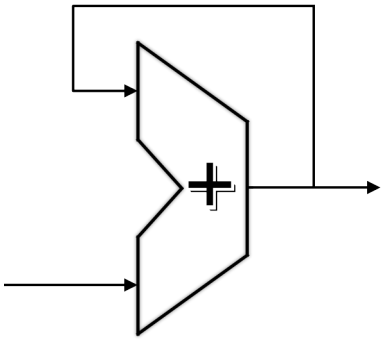
i. Input values are delivered serially, with a data introduction interval(DII) of one cycle,

ii. output order need not match the arrival order of accumulation sets,

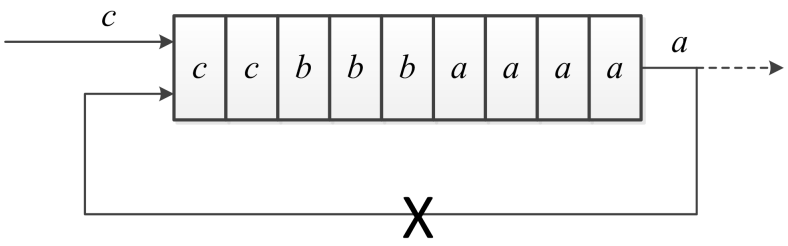
iii. the accumulation sets are contiguous, meaning that the values from different accumulation sets are not intermixed and there is a set ID associated with each incoming value, and

iv. the size of the accumulation sets is not known a priori.

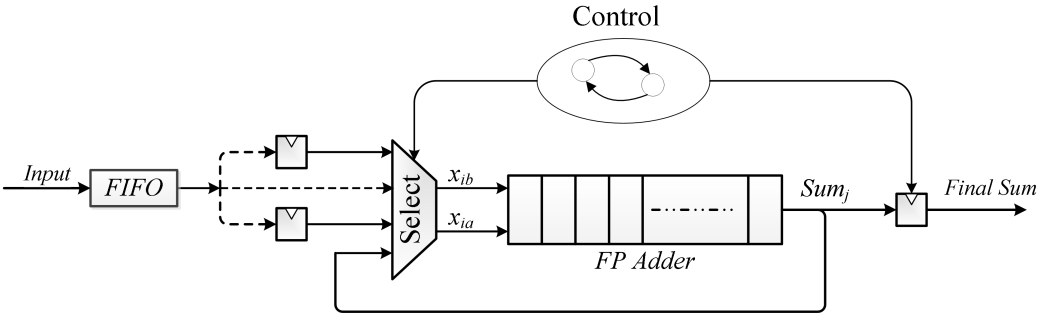
Figure 4.1a shows the design of a simple feedback based accumulator. Since floating point adder is deeply pipelined and the sum is not available in one cycle, there exists a hazard as shown in Figure 4.1b. Values belonging to different set may be added together and hence the results will be wrong. In order to solve this problem, we can allow just one set to be in the accumulator at any point of time until it is reduced completely. With this approach, we will have to use on-chip memory to store



(a) A Simple Accumulator



(b) Feedback based Accumulator with Pipelined Adder



(c) Reduction Circuit with Scheduling

Figure 4.1: Accumulators

other incoming sets. Also, the pipeline will be idle for many cycles while its waiting for the results. Thus, this approach is not only resource intensive but also inefficient.

A special circuit with proper scheduling techniques around floating point adder, as shown in Figure 4.1c is required. In this chapter, we present a reduction circuit which is designed around deeply pipelined double precision floating point adder and addresses the issue of hazard and input scheduling.

4.1 Reduction Circuit Architecuture

The idea behind our reduction circuit design is to expose and exploit parallelism. Pipelining allows multiple additions within an adder to take place simultaneously. We also allow accumulation of multiple sets simultaneously. Thus, we exploit both inter-set and intra-set parallelism. There are various tradeoffs associated with the datapath design around floating point adder. Depth of the floating point adder pipeline affects the design parameters such as complexity of control logic, number of buffers required; i.e., if we increase the depth of the adder, resources-LUTs and registers required on the FPGA- and the multiplexer fan-in increases, and hence the overall performance of the circuit in terms of clock frequency goes down. While designing the reduction circuit we have taken into consideration all these issues.

The reduction circuit has been designed by adding control logic- comparators, counters, and buffers- around a standard deeply pipelined double precision adder in order to form a dynamically scheduled accumulator. In this design, we have combined the input and output buffers and refer them as buffers.

4.1.1 Datapath Components

As shown in Figure 4.2, the reduction circuit architecture can be broadly divided into five interdependent components namely Input, Buffers, Adder pipeline, Counter Mechanism and Control Logic. While all these components work in conjunction,

Datapath Rules in Control Logic govern the functioning of the other components. In this section, we describe each of the components.

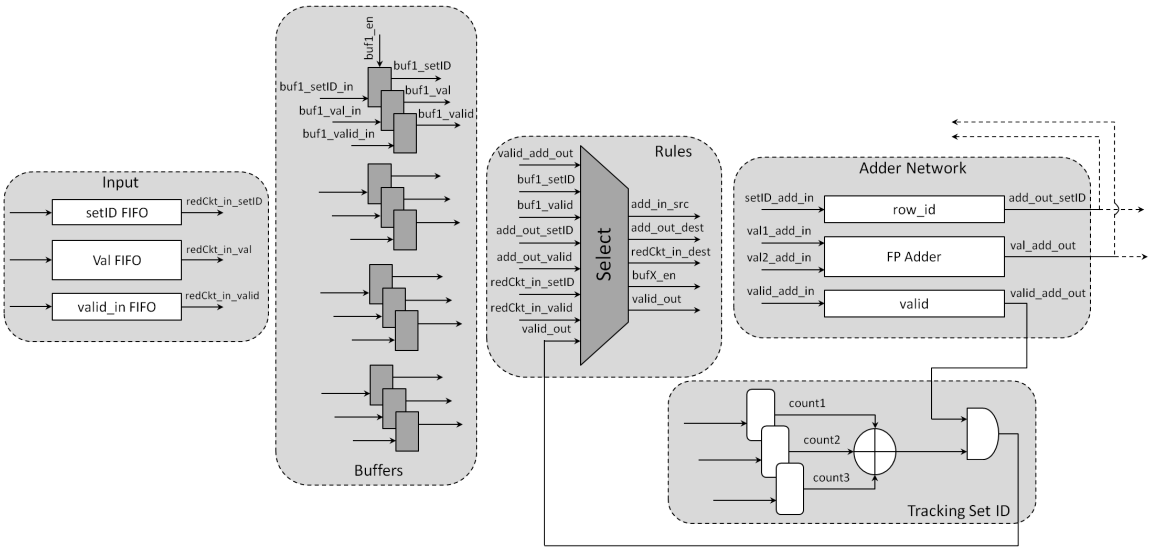


Figure 4.2: Reduction Circuit Components

In order to describe the rules in a more concise manner, we represent the incoming input value to the accumulator as *input.value* and *input.set*, buffer n as $buf_n.value$ and $buf_n.set$, the value emerging from the adder pipeline as *adderOut.value* and *adderOut.set*, the inputs to the adder pipeline $addIn_1$ and $addIn_2$ and the reduced accumulated sum as *result.value* and *result.set*. Also, we represent the number of partial sums belonging to set s as *numActive(s)*.

4.1.1.1 Input

The Input component consists of two FIFOs of equal depth- one for the input set ID and the other for input value. The setID FIFO is 32 bit wide while the Val FIFO is 64 bit wide. The write_enable signals for both the FIFOs are enabled or disabled simultaneously by external mechanism depending on the supply of input data. The read_enable signal is controlled by the control unit in the reduction circuit. Thus,

reduction circuit is supplied the input values along with the respective set ID using these FIFOs.

4.1.1.2 Buffers

The Buffer component consists of multiple buffer triplets. Each triplet has three registers- one for set ID (32 bit), one for value (64 bit) and one for valid signal (1 bit) referred to as buffers. The valid buffer signifies whether the value in the Val buffer is valid or not. An invalid value is not considered for reduction and can be overwritten. The buffers are enabled simultaneously for writing. If a value in the triplet is valid, the triplet can be overwritten only after it has been read. Further, if a valid value is not available for writing after reading the triplet, the triplet is invalidated. The $buf_n.in.valid$ signal is set by the control logic while $buf_n.valid$ signal is used in control logic to determine the inputs to the adder where n denotes the triplet number which we refer to buffer number.

4.1.1.3 Adder Pipeline

The Adder Pipeline consists of two delay lines and a double precision floating point adder. The *set* delay line is used for keeping track of the set ID currently in the floating point adder. The valid delay line checks whether the value in the pipeline is valid or not. Thus, the set ID propagate through the delay line while the sum is being calculated. The input to the adder pipeline is decided by Datapath Rules while the output set ID is used for determining the course of next inputs. The depth of delay lines is equal to the depth of floating point adder.

4.1.1.4 Datapath Rules

The reduction circuit consists of a set of data paths that allow input values and the adder output to be delivered into the adder or buffered based on their corresponding

accumulation set ID and the state of the system. Data paths are established by the control unit according to five basic rules as listed below:

Rule 1: Combine the adder output with a buffered value. Buffer the incoming value.

Rule 2: Combine two buffered values. Buffer the incoming value. Buffer the adder output (if necessary).

Rule 3: Combine the incoming value with the adder output.

Rule 4: Combine the incoming value with a buffered value. Buffer the adder output (if necessary).

Rule 5: Combine the incoming value with 0 to the adder pipeline. Buffer the adder output (if necessary).

We modeled this behavior of reduction circuit and tested various input combinations for different pipeline latency in order to find the maximum number of buffers required. We found that for a 14 stage adder, 9 buffers are required such that there is no overflow and all the sets are reduced correctly. The number of comparisons required and the fan-in to the adder makes it difficult to place and route the design and hence we use 4 buffers and an input 32 stage deep FIFO in the reduction circuit. In order to reduce the number of buffers, we added a special rule which is applied if all the buffers are occupied and rules 1-5 cannot be applied. The special case is listed below:

Rule 6: Combine the adder output with 0.

Figure 4.3 shows various configurations of the reduction circuit: *a*) the output of the pipeline belongs to the same set as a buffered value; *b*) two buffered values belong to the same set *c*) the incoming value and adder output belong to the same

set; *d*) the incoming value and buffered value belong to the same set; *e*) the incoming value does not match the set of the pipeline output or any of the buffered values; *f*) all the buffers are occupied. Algorithm 5 describes the datapath rules.

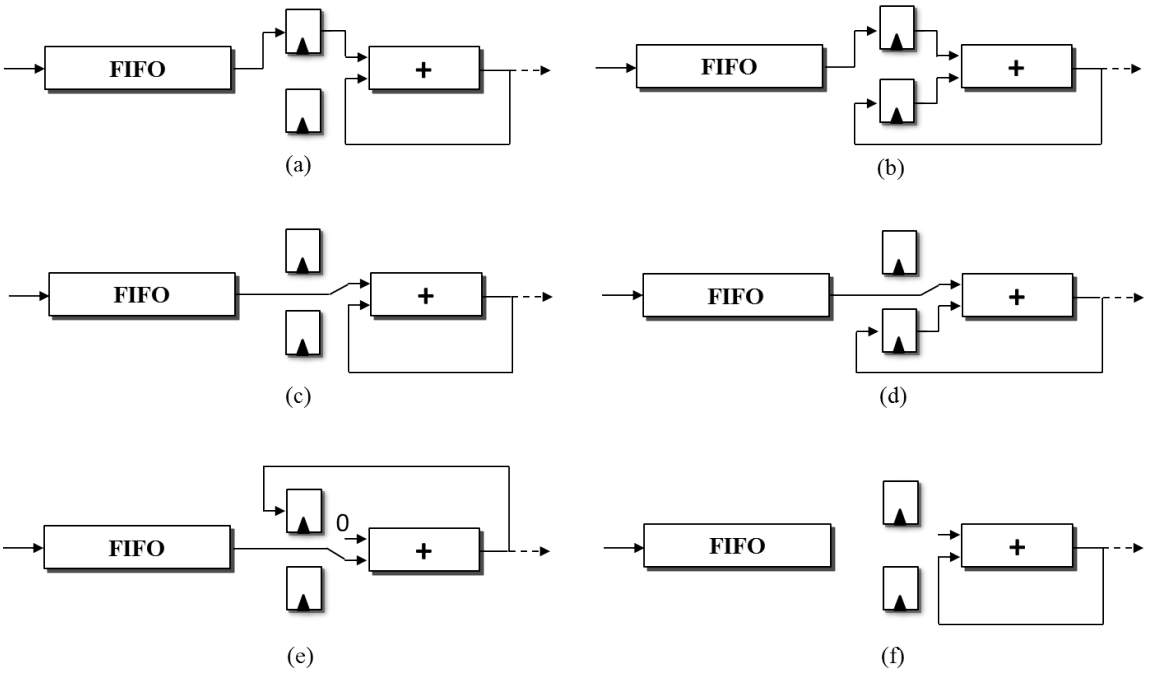


Figure 4.3: Reduction Circuit Rules

4.1.1.5 Reduction Status of Sets

In order to know when a set has been reduced completely, the entries associated with the sets must be tracked. Since the number of entries per set is not known a priori and multiple sets undergo reduction at the same time, we use the corresponding set ID to track of the entries. We have implemented a counting mechanism which notifies the control logic if the set coming out of the adder has been reduced completely.

As shown in Figure 4.4, we use three small dual-ported memories, each with a corresponding counter connected to the write port, in order to determine when a set ID has been reduced (accumulated) into a single value. Together, these memories keep track of the number of active values belonging to each set ID in each cycle, i.e.

Algorithm 5 Reduction Circuit Rules

```
1: if ∃n : buf_n.set = adderOut.set then                          ▷ Rule 1
2:     addIn_1 = adderOut
3:     addIn_2 = buf_n
4:     if input.valid then
5:         buf_n = input
6:     end if
7: else if ∃i, j : buf_i.set = buf_j.set then                     ▷ Rule 2
8:     addIn_1 = buf_i
9:     addIn_2 = buf_j
10:    if input.valid then
11:        buf_i = input
12:    end if
13:    if numActive(adderOut.set) = 1 then
14:        result = adderOut
15:    else
16:        buf_i = adderOut
17:    end if
18: else if input.valid then                                      ▷ Rule 3
19:    if input.set = adderOut.set then
20:        addIn_1 = input
21:        addIn_2 = adderOut
22:    end if
23: else if input.valid then                                      ▷ Rule 4
24:    if ∃n : buf_n.set = input.set then
25:        addIn_1 = input
26:        addIn_2 = buf_n
27:        if numActive(adderOut.set) = 1 then
28:            result = adderOut
29:        else
30:            buf_n = adderOut
31:        end if
32:    end if
33: else if input.valid then                                      ▷ Rule 5
34:    addIn_1 = input
35:    addIn_2 = 0
36:    if numActive(adderOut.set) = 1 then
37:        result = adderOut
38:    else
39:        if ∃n : buf_n.valid = 0 then
40:            buf_n = adderOut
41:        else
42:            ERROR
43:        end if
44:    end if
45: else                                                          ▷ Rule 6
46:    addIn_1 = AdderOut
47:    addIn_2 = 0
48: end if
```

numActive(). These memories cannot be reset once the reduction circuit has been activated.

Note that these memories must contain at least n locations, where n is the maximum possible number of active sets, in order to recycle locations in these memories. At this time each memory has a depth of 256, which we experimentally verified to be sufficient for all the datasets that we have tested. Thus, we use the least significant 8 bits of the set ID as input to each of the memories.

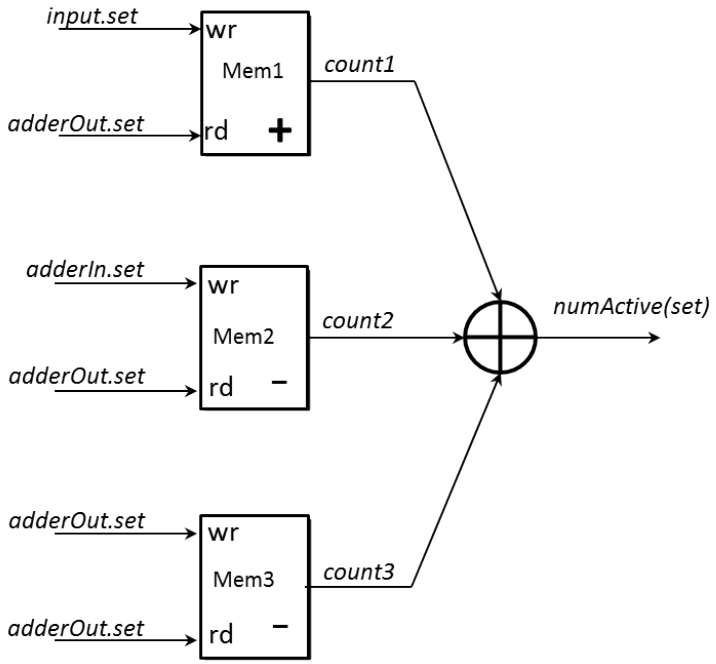


Figure 4.4: Tracking Set ID

The write port of each memory is used to increment or decrement the current value in the corresponding memory location. The write port of one memory is connected to input.set and always increments the value associated with this set ID corresponding to the incoming value. Thus, whenever a value belonging to a particular set arrives, this counter is enabled.

The write port of the second memory is connected to adderIn.set and always decrements the value associated with this set ID whenever two values from this set

enter the adder. This occurs under all rules except for 5, since each of these rules implement a reduction operation. Hence, whenever two valid values enter the input pipeline, this counter is enabled.

As said before, we use the least significant 8 bits of the set ID and the memories used in the counters cannot be reset and hence, after every 256 set IDs, a memory location will be reused while the old counter values are still there. In order to determine correct reduction, we require the third counter. In the third counter, the write port of the third memory is connected to adderOut.set and always decrements the value associated with this set ID whenever the number of active values for this set ID reaches one. In other words, this counter is used to decrement the number of active values for a set at the time when the set is reduced to single value and subsequently ejected from the reduction circuit.

The read port of each memory is connected to *adderOut.set*, and outputs the current counter value for the set ID that is currently at the output of the adder. These three values are added to produce the actual number of active values for this set ID. When the sum is one, the controller signals that the set ID has been completely reduced. When this occurs, the set ID and corresponding sum is output from the reduction circuit.

Figure 4.5 shows the reduction circuit module. Apart from the clock and reset signal, the inputs to the reduction circuit are value (*input*), set ID (set_id_{in}), *valid_in* and an enable (*en*) signal. The *valid_in* signal denotes whether the input is valid. When the *en* signal is de-asserted, the reduction circuit is disabled and the state is preserved. This signal facilitates disabling the reduction circuit when the input stream is discontinued. The outputs of this architecture are the summation of dataset (*output*), set ID (set_id_{out}) and a *valid_out* signal which denotes whether the output is valid or not.

In this chapter, we presented a novel streaming reduction circuit for set-wise

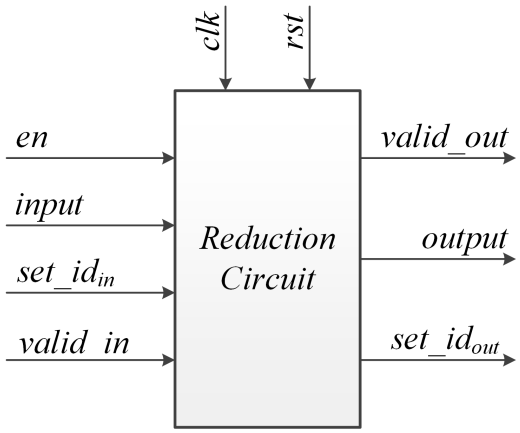


Figure 4.5: Reduction Circuit Module

floating point dataset accumulation. This is designed around a deeply pipelined floating point adder. The inputs to the adder are scheduled dynamically according to the rules we defined earlier. In this reduction circuit, apart from the adder, an input FIFO, 4 buffers, 3 counters and simple control logic is required. Also, the resources are utilized efficiently as the adder does not remain idle. We discuss the performance aspects- operating frequency and resource requirements- of this design in Chapter 6.

Chapter 5

Research Implementation

As discussed previously, in order to improve the accuracy of summation, we may need to reorder the input values. These methods are not suitable for high performance implementations as sorting algorithms are compute intensive and require prior knowledge of datasets. Compensated summation methods provide another avenue for improvement of accuracy but require additional floating point operations and comparisons between two input values to extract error. Accuracy can also be improved by performing the intermediate operations in extended precision but wider thus deeper adder and wider buffers to store intermediate results are required. Thus all the methods to improve accuracy come at the cost of increased number of floating point operations and complexity. But compensated summation methods and intermediate operations in extended precision do not require upfront processing and a priori knowledge of dataset, these are suitable for high performance implementations on FPGAs.

Compensated summation when implemented in software, require explicit comparison of input values and the values may be swapped. This translates to branch operations and hence more dependencies. Implementing these methods in hardware necessarily increases the number of resources required. Also, the control logic to resolve dependencies may become more complex which can result in slower operating frequency. As such, the number of floating point operations per unit of time may be more than that for simple recursive summation but this does not translate to improved throughput as the number of operations per result is more hence overall

performance degrades.

While making a choice for implementation in hardware, we must consider factors such as resource requirement and complexity of control logic. For high performance applications on FPGAs, techniques which require reordering the input data are not practical as the upfront cost of sorting is very high. Further, these methods require more on-chip memory. Lack of sufficient on-chip memory increases the off-chip communication which itself is a bottleneck.

Methods in which the errors are calculated and incorporated in the results i.e. compensated summation methods are comparatively less expensive. Utilizing extended precision in intermediate operations for improving accuracy is also an attractive option.

In this dissertation, we present a set-wise floating point accumulation framework for FPGAs which not only reduces multiple streaming sets efficiently but also improves the accuracy of the results. The objective of this is to evaluate various design tradeoffs such as resource usage and working frequency for different methods. Our goal is to achieve high throughput while maintaining high accuracy and keeping the resource requirement low. The reduction circuit architecture described in the previous chapter serves as the foundation for this framework.

The designs we have implemented for improving the accuracy of the results can be categorized in two approaches. Firstly, we present two designs based on compensated summation. As mentioned earlier, compensated summation methods require additional floating point operation to extract the rounding error. In order to eliminate the additional operations for extracting the error, we have designed a custom floating point adder which along with the sum of two floating point numbers outputs the error as well. This error is incorporated in the result.

In the second approach, we use an extended precision adder in the current reduction circuit design. In this implementation, the inputs are converted to extended

precision values and all the intermediate calculations are performed in extended precision.

These designs have been designed targeting Xilinx Viretx 5 LX330 FPGA but can be ported to other FPGAs as well without any significant changes. In order to verify the results of the architectures, we to compare the results with the results from software model of Kahan's compensated summation method and software version of recursive summation with extended precision using MPFR library. We also report the resource usage and working frequency for an Xilinx Viretx 5 LX330 FPGA.

In the following sections, we discuss the implementation details of the custom floating point adder, the two designs based on compensated summation and the design based on extended precision.

5.1 Custom Floating Point Adder with Error Output

Compensated summation algorithms such as those based on Fast2Sum provide more accurate result for summation but require additional floating point operations. These may seem attractive for implementation in software but when implementing these in hardware with standard floating point units, the overall latency of the reduction operation increases hence the resource requirements and complexity of control logic governing the inputs also increases and hence the performance is adversely affected.

Figure 5.1 shows the adder network for hardware implementation of the Fast2Sum algorithm using standard double precision adders for extracting the error during addition. Here, the order of operations requires the comparison between the absolute values of input operands a and b. If $|a| < |b|$ then we swap these values and supply them to the adder network. This comparison is in effect the comparison between the exponents of operands and during addition, the operand with smaller exponent is shifted. The adder network consists of three double precision adders and output of the third adder is the required error output. Thus, the latency of Fast2Sum

algorithm, when implemented in hardware is more than three times the depth of an adder including the cycles required to compare the two inputs. Though, this approach is simple as standard adders can be used but the resource requirement is quite high. Also, if we were to use this approach along with the reduction circuit architecture for set-wise accumulation, the number of buffers required would have been high. Also, we would require a deeper input FIFO. Due to an increase in the number of buffers, more comparisons would be required. The control logic would become more complex and slow.

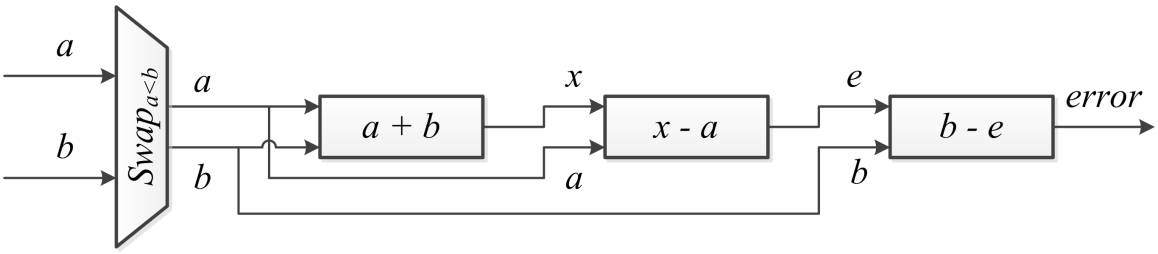


Figure 5.1: Fast2Sum Algorithm in Hardware

Our goal is to implement compensated summation algorithms such that the overall latency and the resource requirement of the reduction circuit is optimal.

As described previously, compensated summation methods recover the rounding error which may consists of the shifted out portion of smaller operand or the difference between the rounding and the shifted out bits. We have designed a custom double precision floating point adder which takes two input values and outputs the error encountered during the addition along with the sum of two input values. We used the double precision floating point adder generated using FloPoCo[41] framework. Instead of using three adders for recovering the error, we can just use one adder and thus, the latency and resource requirements are greatly reduced.

Here, we describe the design of a custom double precision adder, which takes into consideration the shifted out bits of the smaller operand and the rounding direction

of the final result on the basis of which computes and outputs the appropriate error term.

5.1.1 Error Extraction

The rounding error in summation is constituted either by the shifted out bits of the smaller operand or the difference between round value and shifted out bits. The error depends on the direction of rounding and sign of the operands. Thus, for calculating the error we need to know whether the result was rounded up or down and whether the sign of operands are same or different. There are four possible cases:

Case 1: a and b have same sign, result is rounded down: In this case, b is shifted right for alignment and effective addition is performed. Shifted-out bits of b, *sh(b)* with exponent of a form the error term in unnormalized form. In order to normalize it, we need to find the leading number of zeros and shift the bits to left by the number of zeros and subtract the leading zero count from exponent of a. The sign of the error is same as sign of a. The error is added to the final result for compensation as the rounding operation caused the sum to be less than the exact value.

If during the normalization of addition of a and b, right shift operation is performed, the shifted out bit becomes the most significant bit of the error term and the exponent of the adder is adjusted appropriately.

Case 2: a and b have same sign, result is rounded up: In this case, b is shifted right for alignment and effective addition is performed. Since the result is rounded up, the error term is derived from subtraction of shifted out bits from round bit. More precisely, the error is equal to $2^{exp(a)-t} - sh(b)$. Sicne round up caused the sum to be more than the exact value, the sign of the error is opposite that of sign of a. The

error is subtracted from the final result for compensation. Here, t is the number of bits in the mantissa and $t = p - 1$ with precision p.

Case 3: a and b have different sign, result is rounded down: In this case, b is shifted right for alignment and effective subtraction is performed. The error term is calculated as $(2^{exp(a)-t} - sh(b))$. Sign of error is same as the sign of a. The error is added to the final result for compensation as the rounding operation caused the sum to be less than the exact value.

Case 4: a and b have different sign, result is rounded up: Here, b is shifted to right and effective subtraction takes place. The error term is equal to $2^{exp(a)-t} - (2^{exp(a)-t} - sh(b))$ i.e. $sh(b)$. The sign of the error is opposite that of a. The error is subtracted from the final result for compensation as the rounding operation caused the sum to be more than the exact value.

All these cases have been illustrated in Figure 5.2 with the help of examples. For the sake of simplicity, we have used less number of bits but similar procedure is followed for double precision format. During addition, the mantissas of input operands are stored in wider registers. The extra bits consist of the implied 1, one bit for carry bit and guard, round and sticky bits. Thus, for double precision, we use 56 bits during intermediate addition. Rounding depends on the least significant bit (LSB) of mantissa, and guard, round and sticky bits, which are highlighted in the figure.

5.1.2 Custom Hardware Implementation

In order to design a custom adder for rounding error extraction, we need to consider all the above described cases. In the adder design, the algorithm for floating point addition described previously is followed but in order to add the error output functionality and cover all the cases, we have made modifications to the design. Figure 5.3

Case 1: Input operands A and B have same sign, result is rounded down.

$A = 1.1001110110 \times 2^5 = 1.1001110110000 \times 2^5$

$B = 1.1101011011 \times 2^2 = 0.0011101011011 \times 2^5$

$S = 1.1101100001011 \times 2^5 \approx 1.1101100001 \times 2^5$ (Round Down)

$Error = 0.0000000000011 \times 2^5 = 1.1 \times 2^{-7}$

Case 2: Input operands A and B have opposite sign, result is rounded up.

$A = 1.1001110110 \times 2^5 = 1.1001110110000 \times 2^5$

$B = 1.1101011111 \times 2^2 = 0.0011101011111 \times 2^5$

$S = 1.1101100001111 \times 2^5 \approx 1.1101100010 \times 2^5$

$Error = -(0.0000000001000 \times 2^5 - 0.0000000000111 \times 2^5$

$= -0.0000000000001 \times 2^5 = -1.0 \times 2^{-8}$

Case 3: Input operands A and B have different signs, result is rounded down.

$A = 1.1001110111 \times 2^5 = 1.1001110111000 \times 2^5$

$B = -1.1101010101 \times 2^5 = -0.0011101010101 \times 2^5$

$S = 1.0110001100011 \times 2^5 \approx 1.0110001100 \times 2^5$ (Round Down)

$Error = 0.0000000001000 \times 2^5 - 0.0000000000101 \times 2^5$

$= 0.0000000000011 \times 2^5 - 1.1 \times 2^{-7}$

Case 4: Input operands A and B have different signs, result is rounded up.

$A = 1.1001110110 \times 2^5 = 1.1001110110000 \times 2^5$

$B = -1.1101011011 \times 2^2 = -0.0011101011011 \times 2^5$

$S = 1.0110001010101 \times 2^5 \approx 1.0110001010 \times 2^5$

$Error = -(0.000000001000 \times 2^5 - (0.0000000001000 \times 2^5 - 0.0000000000101 \times 2^5))$

$= -0.0000000000101 \times 2^5 = -1.1 \times 2^{-7}$

Highlighted bits are L, G, R and S bits which are used for deciding the rounding direction.

Figure 5.2: Rounding Error due to Round and Shift

depicts this new adder which has additional components. To be more specific, an integer subtractor, two leading zero counter and shifters and logic to predict rounding, preserve shifted out bits of smaller operand and select the appropriate value as error is required.

In the first stage of this adder, the exponents of the two input values, a and b, are compared and if $exp(a) < exp(b)$, then the values are swapped. Signal SignX is

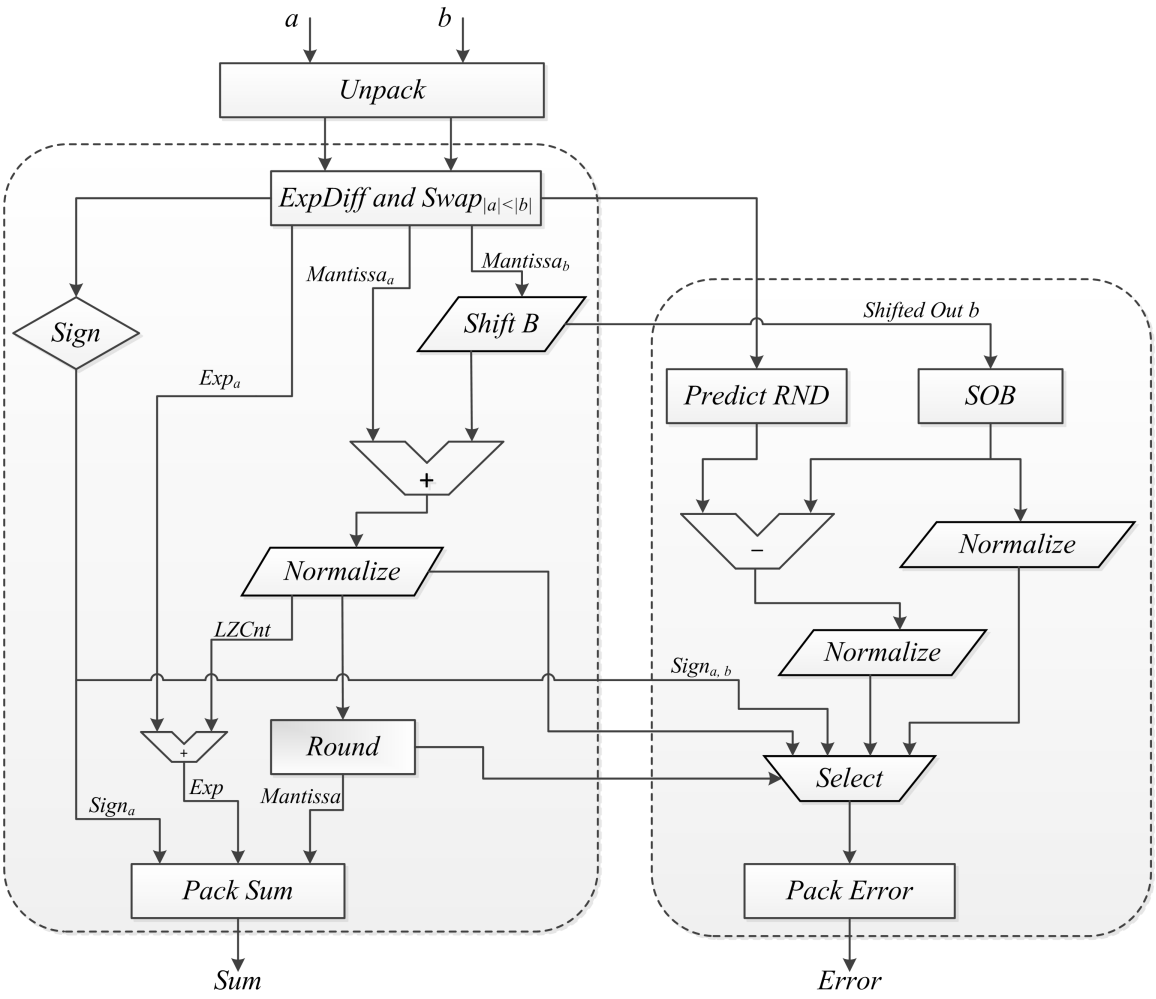


Figure 5.3: Custom Floating Point Adder with Error Output

assigned the sign of the input with greater exponent. Also, the difference between the exponents, *expdiff* is calculated.

In the second stage, the smaller of the two values is shifted to the right by *expdiff*. The shifted out bits (SOB) are preserved along with the exponent of the smaller value if *expdiff* is less than 52. Since we are targeting FPGAs and designing custom hardware, we have the flexibility of using wider registers. Thus, all the bits which have been shifted out can be preserved but if the difference between the exponents is greater than 52, all the bits of b are effectively shifted out. In such a case, we do not consider the shifted out bits for error. SOB is not normalized at this stage.

Normalization and subtraction used for calculating rounding error in later stages take multiple cycles. In order to limit the latency of the error outputting adder to that of original adder, we anticipate the round operation. In order to achieve this, we set a bit at *expdiff*'s place in another register, PREDICT_RND.

In the subsequent stage, difference between PREDICT_RND and SOB, RND_ERR is calculated. This result is normalized using a leading zero counter and shifter unit (LZOC_shift). Also, SOB is normalized and is termed as SHIFT_ERR. Depending on whether the final result is rounded or not and the signs of a and b, one of RND_ERR and SHIFT_ERR with necessary calculations is chosen as the final error.

In this design, we make use of two addition LZOC_Shift units one of which is used for normalization of shifted out bits and the other one is used for normalization of anticipated rounding result. We anticipate rounding and perform the error calculations in accordance with this. This is done because rounding is performed towards the end of the addition process. Using the anticipation approach, the error is available for output along with the sum of input values. Thus the latency of the adder remains the same. If we were to perform error calculations after the rounding decision, the error would have been available several cycles after the sum. This would have essentially increased the overall latency of the the adder.

It can be observed that we require additional resources and logic in the floating point adder design. This not only leads to increased resource utilization but also lowers the operating frequency of the adder. We elaborate and discuss the performance and resource requirements in the next chapter.

In order to incorporate the error during floating point summation in the designs, we use this custom adder in the following designs where we implement compensated summation methods. Using this adder essentially reduces the number of floating point operations in the algorithms and hence reduces the overall latency of the adder network in our designs.

5.2 Accumulated Error Compensation

Using this custom adder, we implement an algorithm in which the errors during addition of values is calculated. The errors are accumulated. Once the set has been reduced completely, we add the accumulated error to the summation value of the set. We call this approach *accumulated error compensation(AEC)*.

Though this approach may seem similar to the one in algorithm 3, the major difference is the order of addition of values. In algorithm 3, recursive summation is performed but the reduction circuit does not necessarily perform recursive summation as it generates partial sums which can be added together during the course of reduction. Thus the order of summation and error generation differs.

Figure 5.4 shows the architectural overview of AEC. In this design, two reduction circuits are required. First reduction circuit is for the summation of dataset values. This reduction circuit also outputs the errors as the adder used in this is the custom adder. We call this *Value Reduction Circuit* (VRC). The second reduction circuit accumulates the errors being generated from the first reduction circuit. We call this *Error Reduction Circuit* (ERC). Once the set is completely reduced, the accumulated error can be added to the final sum using another floating point adder. It must be noted that ERC also performs set-wise accumulation of error values and error values belonging to different sets are not added together.

In VRC, the floating point adder has been replaced by the error outputting custom adder which was described in the previous section. VRC gets the inputs from the input stream while the errors generated by the adder in VRC make the input stream for ERC.

In order to accumulate the error term generated by the adder in the reduction circuit, we need another reduction circuit- error reduction circuit (ERC). Though, ERC is based on the same set of rules as used for the reduction circuit but we need to modify the architecture so that the errors are reduced correctly. This is due to

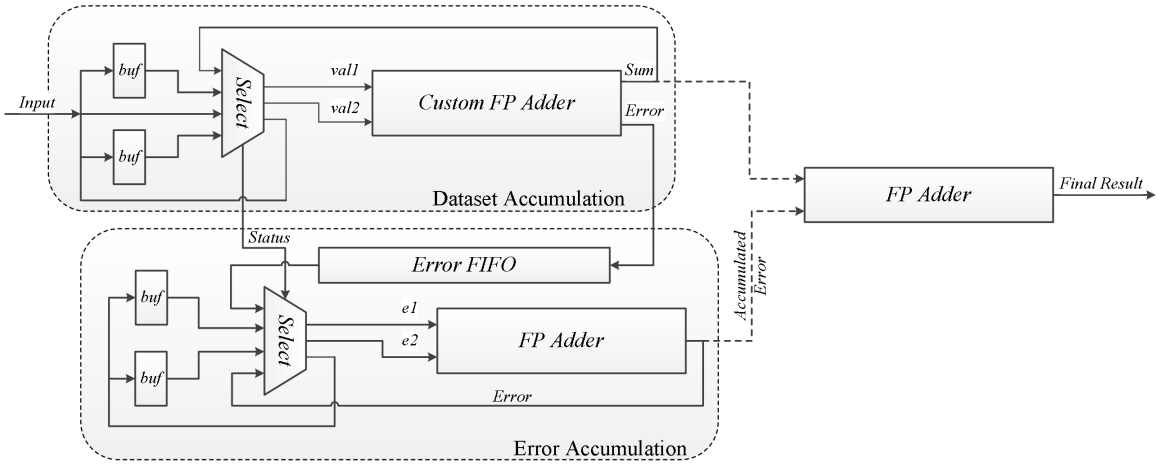


Figure 5.4: Error Accumulation and Compensation

the difference in data introduction interval in VRC and ERC. In VRC, the sets are delivered contiguously but at the same time, multiple sets are reduced simultaneously. Consecutive outputs from the adder may belong to different sets. Hence, ERC does not necessarily receive the values belonging to the same set in contiguous manner. Because of this, there may arrive a situation where the valid flag in ERC is raised, signifying a set reduction for errors, while the main set is still being reduced in VRC. Also, when Rule 5 and 6 are applied in VRC, the output error values are 0, but if supplied to ERC, the number of error terms to be reduced will be greater than the main set size. Though 0's do not affect the functionality of ERC but lead to additional floating point operations hence poor throughput.

We have made modifications to the original reduction circuit architecture in order to accommodate these situations. Figure 5.5 shows the modified module for the VRC and gives an overview of ERC.

In VRC, we have added logic such that it supplies error terms outputted from the adder to ERC only when the adder output is valid and rule 5 or rule 6 were not applied. The valid signal is already present in the original reduction circuit. We have added a delay line whose input is a signal rule_5_or_6 and is asserted when rule 5

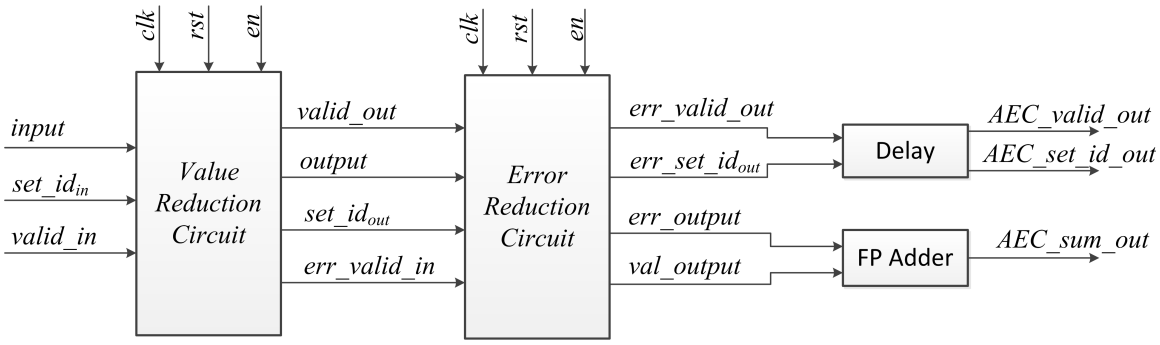


Figure 5.5: Module for Error Accumulation and Compensation

or 6 is applied. The depth of this delay line is equal to the depth of the adder pipeline which is 14 stages. Thus, the valid signal for the error term can be described using the following equation:

$$err.valid = adderOut.valid \text{ and not(adderOut.rule\_5\_or\_6)} \tag{5.1}$$

It must be noted that the overall behavior of VRC remains the same as the original reduction circuit. Also, the number of errors to be reduced by ERC is one less than the total number of values in the corresponding dataset.

In ERC, we need to add mechanism such that it is able to deal with non-contiguous supply of data and is still able to reduce the errors correctly. In order to deal with this situation, the status of set in VRC needs to be checked whether it has been completely reduced. This can be checked using the valid_out signal when an error term is supplied to ERC. If the valid_out signal is asserted, then the main set has been completely reduced and the last error value has been supplied to ERC. This signal is synchronized with the input FIFO in ERC and is stored in a dual ported memory in ERC. In order to assert valid_out_err in ERC, this signal must be asserted and the sum of outputs of the three counters must be 1. Due to this additional check behavior of other components namely the control logic including the rules and the counters does not change in ERC. In ERC, we need to add mechanism such that it is able to deal with non-contiguous supply of data and is still able to reduce the

errors correctly. In order to deal with this situation, the status of set in VRC needs to be checked whether it has been completely reduced. This can be checked using the valid_out signal when an error term is supplied to ERC. If the valid_out signal is asserted, then the main set has been completely reduced and the last error value has been supplied to ERC. This signal is synchronized with the input FIFO in ERC and is stored in a dual ported memory in ERC. In order to assert valid_out_err in ERC, this signal must be asserted and the sum of outputs of the three counters must be 1. Due to this additional check behavior of other components namely the control logic including the rules and the counters does not change in ERC.

Also, the final summation of the main set from VRC is stored in another dual ported memory in ERC as the final error cannot be available immediately after the set reduction. Once the error terms belonging to a set have been reduced, the final sum is calculated by applying the error correction by adding it to the error term from the adder and main set sum from the memory. For this we need another floating point adder.

Thus, in AEC we emulate compensated summation algorithm in hardware without requiring the comparison between the inputs and explicit floating point hardware for calculating the rounding error. Also, since the error extraction is integrated within the floating point adder, the pipeline depth of VRC and ERC is same as the original reduction circuit.

We discuss the accuracy results, resource requirements and the performance in terms of operating frequency of this design in the next chapter.

5.3 Adaptive Error Compensation in Subsequent Addition

We have implemented a modification to the approach described in algorithm 1. We refer to this approach as *adaptive error compensation in subsequent addition(AECSA)*. In this design, we calculate the errors during summation and if possible apply a cor-

rection as well. If it is not possible to apply the correction because of unavailability of corresponding error, we accumulate the error. Once the set has been reduced completely, we add the final accumulated error term and the result. This approach has been depicted in Figure 5.7.

Figure 5.6 shows the pipeline required for implementing one iteration of compensated summation algorithm which compensates for the error in subsequent addition. In this implementation, error term, e calculated in the previous iteration is required as an input to the first adder. Due to the depth of the adder network, it is not possible to have this error every cycle. Error will not be available for each addition even if we replace the two adders with one custom adder which generates error. The values from different sets can be scheduled in an interleaved manner but this will require lot of on-chip resources, complex control logic and upfront processing of data.

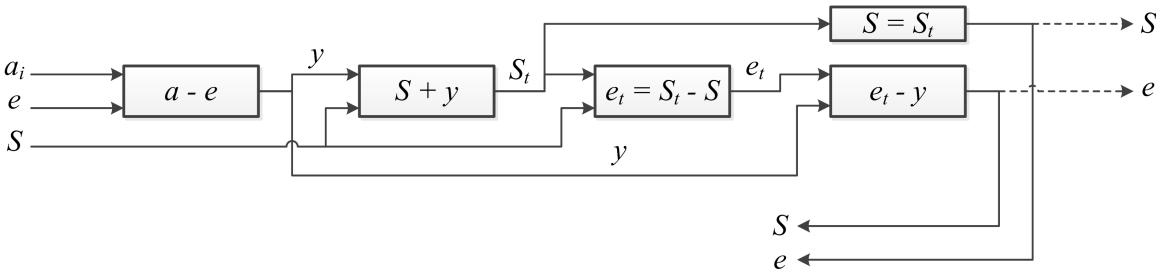


Figure 5.6: Error Compensation in Subsequent Addition

Figure 5.8 shows an overview of AECSA. In this design, we require two reduction circuits but unlike AEC, the application of rules in ERC depends on the availability of errors to VRC. Further, since the error compensation may take during accumulation of values, two adders connected sequentially are required in VRC. The first adder is for error correction which adds the error to an input value while the other adder adds the output of the first adder with the second input value. The second adder also generates the error term.

A comparator at the input of the adder pipeline is required in order to compare

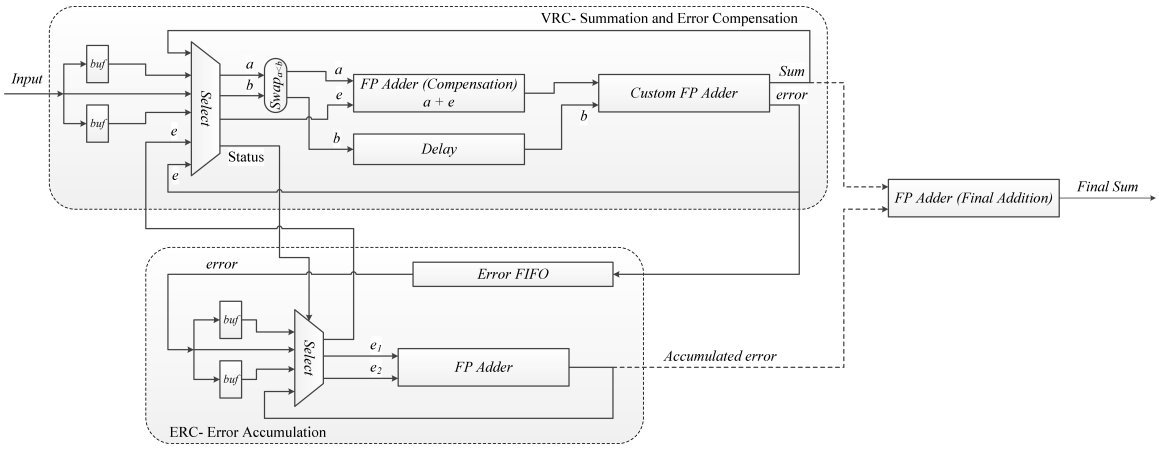


Figure 5.7: Adaptive Error Compensation in Subsequent Addition

the input values. If $|a| > |b|$, a and b are swapped. Thus, the overall adder pipeline depth in VRC is more than two times of that in AEC and original reduction circuit. Here, we increase the size of the input FIFO but keep the number of buffers same as the original reduction circuit in order to keep the control logic simple.

As discussed previously, due to the depth of adder pipeline and simultaneous accumulations from the same set, there may arrive a situation where we may not be able to compensate for the error using the first adder. In such a case, we accumulate the error using ERC. Errors can also be supplied from ERC to VRC in order to maximize the chances of error compensation in VRC. After set reduction, the final error term from ERC can be added to summation to obtain the final result.

The major difference between AEC and AECSA is, in AEC the rules in ERC are independent of conditions in VRC while in AECSA, a check for set ID match is performed and if a match is found then that error value is supplied to VRC. In other words, the set ID of error term in ERC is equal to the set ID which is being input the adder pipeline in VRC, the corresponding value from ERC is supplied to VRC for error compensation.

In ERC, if conditions for a rule are satisfied but the set ID of error source matches in VRC then that rule is not applied. In such a case, some other rule is applied. Error

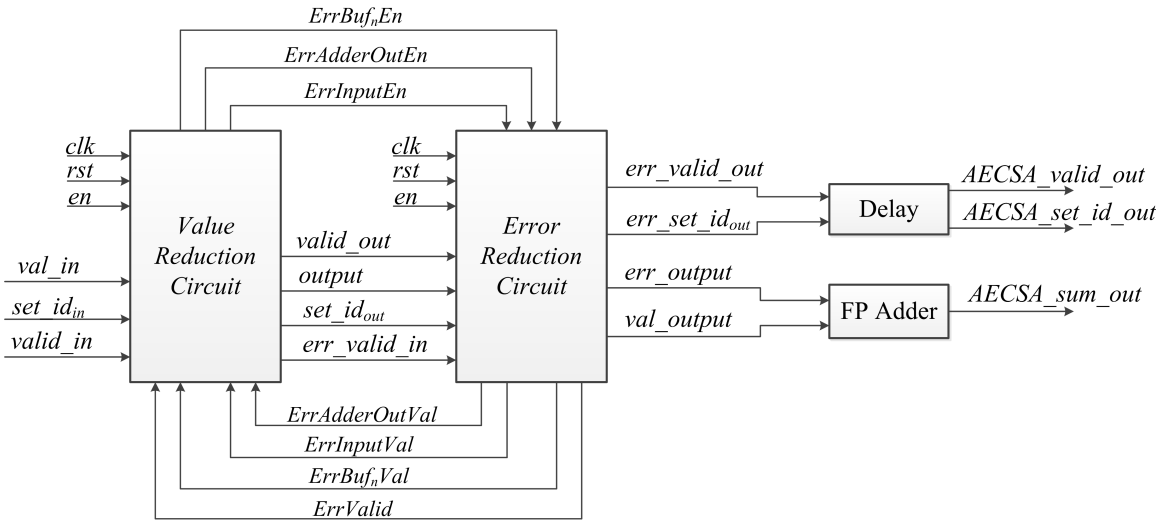


Figure 5.8: Module for AECSA

accumulation in ERC is performed only when is it not possible to supply the errors to VRC as input. Thus, error input to VRC is prioritized. The rules in ERC have been modified in accordance with this policy.

There are four sources of the input error in VRC- the output of the custom floating point adder in VRC, FIFO output in ERC, buffers in ERC and output of adder in ERC. When applying Rule 1, Rule 2, Rule 3 and Rule 4, the set ID of the error outputs is checked compared with the adder input set ID. If a match is found, the respective error term is supplied as input. If the error output from the custom adder in VRC serves as the input error, this error term is not supplied to ERC and err_valid line is de-asserted. If the error term comes from ERC, it is invalidated in ERC and is not considered for accumulation. The rule to be applied in ERC depends on whether the corresponding error term has been invalidated or not. For example, in ERC, if the set ID of adder output matches the set ID of one of the buffers (Rule 1), but at the same time, in VRC, the set ID is the same as set ID of adder output in ERC, then Rule 1 won't be applied in ERC and the error term from output of the adder pipeline in ERC is supplied to VRC. In such a case, some other rule is applied in

ERC. Algorithm 6 describes the control logic in VRC. It is evident that the number of compare operations per cycle is significantly more and the application of a rule in ERC is dependent on the logic decision from VRC. This essentially levies a timing challenge and the performance in terms of operating is affected adversely due to this.

Also, the rules in ERC have been modified to accommodate the condition for potential set ID match in VRC. The control logic for ERC is shown in algorithm 7.

In the original reduction circuit, we have three counters- the first counts up when a new value arrives in the reduction circuit, the second counts down when two values are supplied to the adder while the third counter counts down when a set has been reduced. Since ERC also supplies error terms back to VRC, the number of valid error term decreases even when there is no reduction. In order to account for the error terms supplied to VRC from ERC and keep track of the number of errors belonging to a particular set in ERC, we need a fourth counter. This counter counts down whenever an error term is supplied to VRC from ERC. In the absence of this counter, the error reduction will not be correct. Example in Figure 5.9 depicts the working of counters in ERC. The subscripts in the example represent the counter which is activated. Thus, when two error terms are added in ERC, counter 1 is activated. Similarly, if a value is supplied from ERC to VRC from either the input FIFO, buffers or output of adder, counter 4 is activated.

In ERC, similar to AEC, the error values belonging to a particular set are not contiguous hence we need to check the status of the set VRC. This can be checked using the valid_out signal when an error term is supplied to ERC. If valid_out signal is asserted, then the main set has been completely reduced and the last error value has been supplied to ERC. This signal is synchronized with the input FIFO in ERC and is stored in a dual ported memory in ERC. In order to assert valid_out_err in ERC, this signal must be asserted and the sum of outputs of the three counters must be 1.

Algorithm 6 AECSA VRC Rules

```
1: if ∃n : buf_n.set = adderOut.set then                          ▷ Rule 1
2:     rule1 = 1
3:     addIn_1 = adderOut
4:     addIn_2 = buf_n
5:     if input.valid then
6:         buf_n = input
7:     end if
8: else if ∃i, j : buf_i.set = buf_j.set then                     ▷ Rule 2
9:     rule2 = 1
10:    addIn_1 = buf_i
11:    addIn_2 = buf_j
12:    if input.valid then
13:        buf_i = input
14:    end if
15:    if numActive(adderOut.set) = 1 then
16:        result = adderOut
17:    else
18:        buf_i = adderOut
19:    end if
20: else if input.valid then                                      ▷ Rule 3
21:    if input.set = adderOut.set then
22:        rule3 = 1
23:        addIn_1 = input
24:        addIn_2 = adderOut
25:    end if
26: else if input.valid then                                      ▷ Rule 4
27:    if ∃n : buf_n.set = input.set then
28:        rule4 = 1
29:        addIn_1 = input
30:        addIn_2 = buf_n
31:        if numActive(adderOut.set) = 1 then
32:            result = adderOut
33:        else
34:            buf_n = adderOut
35:        end if
36:    end if
37: else if input.valid then                                      ▷ Rule 5
38:    addIn_1 = input
39:    addIn_2 = 0
40:    if numActive(adderOut.set) = 1 then
41:        result = adderOut
42:    else
43:        if ∃n : buf_n.valid = 0 then
44:            buf_n = adderOut
45:        else
46:            ERROR
47:        end if
48:    end if
49: else                                                          ▷ Rule 6
50:    addIn_1 = AdderOut
51:    addIn_2 = 0
52: end if
```

```
53: if rule1 OR rule2 OR rule3 OR rule4 then
54:     if not(adderOut.rule_5_or_6_out) then
55:         addErrIn = addererrOut
56:         errInput.disable = 1
57:     else if errInput.set = adderOut.set then
58:         addErrIn = errInput
59:         errInput.errEn = 1
60:     else if errAdderOut.set = adderOut.set then
61:         addErrIn = errAdderOut
62:         errAdderOut.errEn = 1
63:     else if ∃n : errBuf_n.set = adderOut.set then
64:         addErrIn = errBuf_n
65:         errBuf_n.errEn = 1
66:     else
67:         addErrIn = 0.0
68:     end if
69: else
70:     addErrIn = 0.0
71: end if
72: if count1 + count2 + count3 = 1 then
73:     redCktOut.valid = 1
74:     redCktOut.set = adderOut.set
75:     redCktOut = adderOut
76: end if
77: if not(errInput.disable)ornot(adderOut.rule_5_or_6_out) then
78:     redCktOut.Err = adderOut.Err
79:     redCktOut.errValid = adderOut.valid
80:     redCktOut.errSet = adderOut.set
81: end if
```

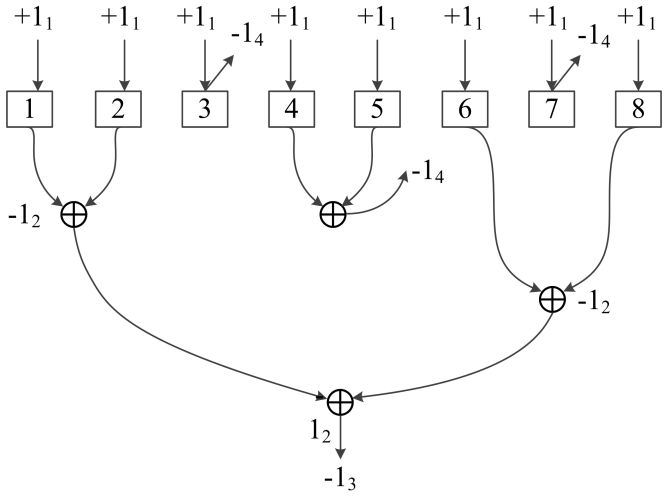


Figure 5.9: Working of Four Counters in AECSA-ERC

Algorithm 7 AECSA ERC Rules

```
1: if ∃n : errBuf_n.set = errAdderOut.set and
2:     not (errBuf_n.errEn or errAdderOut.errEn) then                ▷ Rule 1
3:     rule1 = 1
4:     errAddIn_1 = errAdderOut
5:     errAddIn_2 = errBuf_n
6:     if errInput.valid then
7:         errBuf_n = errInput
8:     end if
9: else if ∃i, j : errBuf_i.set = errBuf_j.set and
10:    not (errBuf_i.errEn or errBuf_j.errEn) then                   ▷ Rule 2
11:    rule2 = 1
12:    errAddIn_1 = errBuf_i
13:    errAddIn_2 = errBuf_j
14:    if errInput.valid then
15:        errBuf_i = errInput
16:    end if
17:    if numActive(adderOut.set) = 1 then
18:        result = errAdderOut
19:    else
20:        buf_i = errAdderOut
21:    end if
22: else if errInput.valid then                                      ▷ Rule 3
23:    if errInput.set = errAdderOut.set and
24:    not (errInput.errEn or errAdderOut.errEn) then
25:        rule3 = 1
26:        addIn_1 = errInput
27:        addIn_2 = errAdderOut
28:    end if
29: else if errInput.valid then                                      ▷ Rule 4
30:    if ∃n : errBuf_n.set = errInput.set and
31:    not (errInput.errEn or errBuf_n.errEn) then
32:        rule4 = 1
33:        errAddIn_1 = errInput
34:        errAddIn_2 = errBuf_n
35:        if numActive(errAdderOut.set) = 1 then
36:            result = errAdderOut
37:        else
38:            buf_n = errAdderOut
39:        end if
40:    end if
41: else if errInput.valid then                                      ▷ Rule 5
42:    errAddIn_1 = errInput
43:    errAddIn_2 = 0
44:    if numActive(errAdderOut.set) = 1 then
45:        result = errAdderOut
46:    else
47:        if ∃n : errBuf_n.valid = 0 then
48:            errBuf_n = errAdderOut
49:        else
50:            ERROR
51:        end if
52:    end if
53: else                                                             ▷ Rule 6
54:    errAddIn_1 = errAdderOut
55:    errAddIn_2 = 0
56: end if
57: if errCount1 + errCount2 + errCount3 + errCount4 = 1 andvrc.set.done = 1 then
58:    errOut.valid = 1
59:    errOut.set = adderOut.set
60:    errOut = adderOut
61: end if
```

Also, the summation of the main set is stored in a dual ported memory. Once the error terms belonging to a set have been reduced, the final sum is calculated by adding the error term from the adder and main set sum from the memory. For this another floating point adder is required.

In AECSA, the overall behavior of VRC does not change. Rules are applied in the original order of priority even when an error term is not available from any of the sources.

5.4 Extended Precision Reduction Circuit

As described in the background section, we can achieve higher accuracy if the working precision of a floating point numbers is increased. For example, double precision generally offers more accuracy than single precision. Also, modern processors, such as Intel's X86 architecture utilize this technique where the floating point operations are performed using 80 bit registers.

In order to implement extended precision, we use a wider adder in the original reduction circuit architecture. We call this design *Extended Precision Reduction Circuit* (EPRC). We have implemented two versions of EPRC, namely EPRC80 and EPRC128. In EPRC80, we use an 80 bit extended precision format while in EPRC128, 128 bit extended precision format has been used.

Figure 5.10 shows the architectural overview of EPRC. The input values are converted to extended precision and are then supplied to the reduction circuit. Thus, we require a wider input FIFO in the reduction circuit. In the reduction circuit, all the intermediate summation operations for calculating the partial sums are carried out using wider floating point adder. The inputs and partial sums are stored in wider buffers. Once a set has been completely reduced, the final sum which is in extended precision is converted back to double precision format.

If we were to use this reduction circuit in some application utilizing double preci-

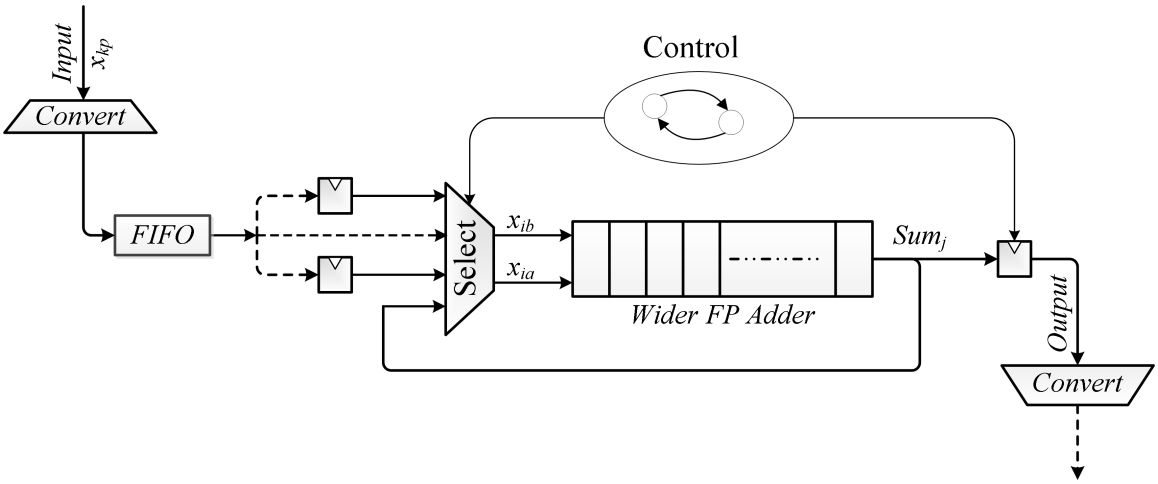


Figure 5.10: Extended Precision Reduction Circuit

sion, the intermediate operations performed in extended precision would be transparent to the application as the outputs are converted back to double precision values.

EPRC offers, though does not guarantee, better accuracy than the original reduction circuit. It may offer better accuracy because less number of bits are discarded during denormalization of smaller operand during intermediate operations. It is generally the case that more number of guard bits are available with a wider adder since the original inputs are in double precision.

Since the outputs are converted back to double precision, the results go through double rounding in EPRC and extended precision arithmetic in general. The first rounding operation is performed during addition while the second is performed during conversion of output to double precision. Though the effect of rounding during addition is mitigated by the use of extended precision as rounding may be required less number of times, but rounding during conversion may be the cause of error. As suggested in the literature, this error cannot be worse than the overall error with double precision summation. Hence, extended precision offers at least as much accuracy as double precision; in general results from operations performed with extended precision are better.

In EPRC, the inputs and outputs are in double precision, hence during the conversion process of inputs, we do not modify the number of exponent bits i.e. the number of exponent bits in the extended precision values is 11. Extra mantissa bits are appended towards the least significant portion of the mantissa. Figure 5.11a shows the 80 bit extended precision used in EPRC80 and Figure 5.11b depicts 128 bit extended precision format used in EPRC128. The width of exponent in both the formats is 11 bits while the mantissa width used in EPRC80 is 68 bits and that in EPRC128 is 116biths. Input and outputs values are in double precision and assume mantissa of 52 bits.

(a) 80 Bit Extended Precision Format for EPRC80

(b) 128 Bit Extended Precision Format for EPRC128

Figure 5.11: Extended Precision Floating Point Format

In double precision addition, only 3 bit shift can be tolerated with guard, round and sticky bits. If exponent difference is larger than 3, shifted out bits are discarded. Extending mantissa to 68 bits can effectively tolerate and preserve 16 least significant bits of the smaller input operand while using 116 bits for mantissa preserves 64 bits. Thus, the error due to shift during the intermediate operations is mitigated completely if the exponent difference is less than or equal to 16 for 80 bit format and 64 for 128 bit format. This does not include the padding offered by guard, round and sticky

bits. The mantissa can be extended further but the depth of the adder also increases with the width of input operands.

An extended precision format necessarily requires wider integer adder and wider shift units which may require additional cycles to effectively pipeline these operations. Thus, the overall latency of a wider floating point adder is more than that of a double precision adder. In EPRC, we use a deeper input FIFO to accommodate a deeper adder. The latency of adder in EPRC80 is 19 cycles while that in EPRC128 is 26 cycles. The overall behavior and rules for both these designs is the same as our original reduction circuit.

5.5 Summary of Designs

In this chapter, we presented a custom double precision adder design for error extraction and three designs for improving the accuracy of set-wise summation. All these designs are based on our reduction circuit. Thus, these exploit inter-set and intra-set parallelism for streaming datasets by allowing accumulation of multiple sets simultaneously.

In Accumulated Error Compensation (AEC), the errors generated by VRC are accumulated in ERC. Once the set is reduced completely in VRC, the accumulated error from ERC is added to the summation to obtain the final result with error correction.

In Adaptive Error Compensation in Subsequent Addition (AECSA), the errors generated during addition can be compensated in VRC using another adder. Thus two adders are required in VRC. ERC can also supply errors to VRC. If it is not possible to consume the error in VRC, they are accumulated in ERC. In order to calculate the final result, the accumulated error from ERC is added to the sum from VRC.

In EPRC80, we use 80 bit while in EPRC128, 128 bit adders is used in our

reduction circuit in order to perform all the intermediate operations in extended precision format. Wider buffers are required to store the intermediate results. The double precision values are converted to 80 bit and 128 bit extended precision values respectively. The final result is converted back to double precision.

Thus, all the designs we presented require additional resources on top of the original reduction circuit. In the next chapter, we discuss the resource requirements, performance and accuracy parameters of these designs in detail.

Chapter 6

Experiments and Results

In this chapter, we discuss resource requirements, performance and accuracy parameters of the new designs with respect to the original reduction circuit.

6.1 Resource Requirements and Operating Frequency

In the previous chapter we presented the design of a custom floating point adder which outputs the roundoff error incurred during floating point addition and four designs for achieving better accuracy for set-wise floating point summation.

In order to characterize the tradeoffs of these designs in terms of operating frequency and resource requirements of the designs, we synthesized, placed and routed the designs using Xilinx ISE 14.4 for the Xlinix Virtex 5 LX330 FPGA. We report the resource usage and frequency from place and route results.

6.1.1 Custom Floating Point Adder

First we discuss and compare the custom floating point adder design. We added necessary components to double precision floating point adder to extract the error which occurs during floating point addition. The error be constituted by the the shifted out bits of the smaller operand or it can be due to the rounding. In order to calculate the error, we need an integer adder (subtractor) which calculates the difference between rounding bits and shifted out bits after necessary appropriations as explained in the previous chapter. Also we need a leading zero bit counter and shifter to normalize the error. Since the rounding decision- whether the result is

rounded up or down- is made towards the end of the addition operation. In order to keep the latency the same as the original floating point adder (14 cycles), we anticipate the round operation and choose the error when rounding decision is made. To facilitate this, we require extra components. Thus, we require two leading zero counter and shifter units, and an integer subtractor for extracting the error.

Table 6.1 shows the resource and frequency comparison between the FloPoCo double precision adder and our custom floating point adder which outputs the error. It can be observed that there is a significant increase in the resource requirements and the resources required are almost twice that of the original design. We attribute this increase to the two leading zero counter and shifter units, the integer subtractor and the logic required to choose the appropriate error term.

It can be seen that the operating frequency is also reduced by 28%. This is due to the additional control logic for error extraction in the floating point adder unit.

Table 6.1: Custom Floating Point Adder

Design	Slice Register	Change	Frequency	%Change
FP Adder	1130		330 MHz	
Custom Adder	2310	2.04X	235 MhZ	-28.7%

6.1.2 Reduction Circuits for Improving Accuracy

The reduction circuit, as described in chapter four, has input FIFOs for set ID (32 bit wide, 32 deep) and input values (64 bit wide, 32 deep), 4 buffer triplets each with one 1-bit register for valid, 32-bit register for set ID and 64-bit register for value and an adder network with a double precision floating point (14 deep), a delay line for valid signal and a delay line for set ID. We described four designs described in chapter 5 for improving the accuracy of set-wise floating point summation which are based on this reduction circuit architecture.

Two of the designs– AEC and AECSA– are based on the principle of compensated summation. In both the designs, two reduction circuits are required. The error is calculated using the custom floating point adder in value reduction circuit (VRC) and is accumulated using the error reduction circuit (ERC). In AEC, ERC acts independently while in AECSA, ERC can supply partially accumulated errors to VRC and the rules in ERC are applied accordingly. Thus, for AEC and AECSA at least twice as many resources than the original reduction circuit are required. Additional logic to accumulate the error properly makes the control logic complex.

In EPRC80 and EPRC128, we use 80 bit and 128 bit extended precision floating point adder with pipeline depth of 19 and 26 respectively. All the intermediate calculations are carried out in the extended precision format. The wider adder and reduction circuit value registers in buffer triplet, the conversion units and a deeper input FIFO (64 deep) increase the device utilization in both these designs.

In this section, we discuss the resource requirements for AEC, AECSA, EPRC80 and EPRC128 and compare it with the original reduction circuit design.

AEC and AECSA are based on the principle error free transformation which forms the basis of compensated summation methods. In both these designs we emulate the compensated summation. But these designs do not implement recursive summation. It is because, in the reduction circuit, addition of two partial sums as well as addition of two input values is possible while in recursive summation, there is only one partial sum which is added to the input value.

Accumulated Error Compensation (AEC)

In AEC, we need two reduction circuits- the value reduction circuit (VRC) accumulates the input values. The custom adder is used for outputting the errors. These errors are supplied to the error reduction circuit (ERC). ERC may receive the errors belonging to a set in non-contiguous manner. In order to reduce the errors correctly,

we have added status memories in ERC. Also, another adder is required for adding the final sum and the accumulated error value. In AEC we require three adders, two of which are standard adders while the adder in VRC is the custom adder. As such, in AEC, we require more than twice the resources than the original reduction circuit. We also require logic and resources for the two reduction circuit and addition memories in ERC for status check. Thus the control logic becomes more complex. Apart from the status check of the set in VRC, the accumulation of errors in ERC does not depend on VRC.

Adaptive Error Compensation in Subsequent Addition (AECSA)

Similar to AEC, in AECSA, we require two reduction circuits and an adder outside those reduction circuits. But unlike AEC, in AECSA, two adders are required in the adder pipeline in VRC. The first adder serves the purpose of error compensation while the second adder, which is custom adder, outputs the error. In order to avoid buffer overflow due to increased latency, the depth of the FIFO in VRC has to be doubled. The errors from ERC can be supplied to VRC for error compensation. In order to do this, the set ID in VRC must be matched against the set IDs in ERC. Since there are six sources of error in ERC– four buffers, input FIFO and output of the adder– the number of comparisons carried out for error in VRC per cycle increases significantly. This manifolds the complexity of control logic. Thus, AECSA utilizes more resources and has more complex control logic than AEC.

Although, in AECSA, error compensation is done in subsequent addition similar to Kahan's compensated summation algorithm but unlike Kahan's algorithm in software, AECSA does not replicate the behavior of recursive summation because of the scheduling. Also, for Kahan's compensated summation algorithm, the error generated in the previous iteration is compensated in the current iteration but in AECSA, as discussed earlier, the error value may not be available for compensation due to the

depth of the adder pipeline and scheduling in the reduction circuit.

Extended Precision Reduction Circuits: EPRC80 and EPRC128

We have also implemented two extended precision reduction circuits- EPRC80 and ERPC128. In these designs, in order to perform all the computations in extended precision, we require wider adders, wider registers in the buffer triplets to store the partial sums and a wider and deeper input value FIFO. In EPRC80, an 80 bit floating point adder having latency of 19 cycles is used while in EPRC128, 128 bit floating point adder with pipeline depth of 26 cycles is used. Due to the increase in pipeline depth, we require deeper FIFO to avoid buffer overflow. Also, we require conversion units for converting the input values from double precision to extended precision and output values from extended precision to double precision. The complexity of control logic remains the same as the original reduction circuit as the set ID width does not change, but the resource requirement increases.

6.1.2.1 Resource Requirements

In this section we discuss the resource requirements for Table 6.2 depicts the number of comparisons for set ID that are required in each of the designs. These are based on the number of comparisons required per rule. Please note that these only include the comparison between set IDs from the input FIFO, buffers and the output of the error and do not include the check for valid signal and other conditions.

Table 6.2: Number of Comparisons for Designs

	Reduction Circuit	AEC		AECSA		EPRC80	EPRC128
		VRC	ERC	VRC	ERC		
Comparisons	15	15	15	41	15	15	15

In the reduction circuit, *Rule 1* requires four comparisons (output of adder compared with four buffers), *Rule 2* requires six comparisons (four buffers), *Rule 3* requires one comparison (input and output of the adder) and *Rule 4* requires four

comparisons (four buffers and input). Thus, 15 comparisons are required in the reduction circuit. Since the behavior of VRC and ERC in AEC is the same as the reduction circuit, the number of comparison for application of rules is 15 in each. Also, the number of buffers do not change in EPRC80 and EPRC128, the number of comparison remains the same. In AECSA, we need six additional comparisons for *Rule 2* and *Rule 4* while seven additional comparisons are required for *Rule 1* and *Rule 3* for error compensation. Thus, 41 comparisons are required in VRC while 15 are required in ERC in AECSA. Thus, the control logic for AECSA is more complex than the other designs.

Table 6.3 shows the resource requirements and operating frequency for AEC, AECSA, EPRC80 and EPRC128. We also compare these with the original reduction circuit. It can be observed that AECSA requires 300% more resources than the original reduction circuit and suffers 28.2% loss in the operating frequency. This can be ascribed to the complex control logic as described previously. EPRC80 requires on 41.8% more resources, while AEC and EPRC128 require almost 150% more resources. The operating frequency of AEC, EPRC80 and EPRC128 is slightly less than the original reduction circuit. Thus all the designs implemented for improving the accuracy of set-wise floating point accumulation require additional resources. Due to increased resources and complexity of the control logic, the operating frequency is affected adversely but the impact is not severe except for AECSA.

Table 6.3: Resource Requirements and Operating Frequency

Design	Slice Registers	% change	Frequency	% change
Reduction Circuit	1873		188.5	
AEC	4743	153.2%	176.2	-6.5%
AECSA	7938	323.8%	135.4	-28.2%
EPRC80	2656	41.8%	181.8	-3.6%
EPRC128	4600	145.6%	182.5	-3.2%

6.2 Accuracy

In this section, we present the numerical accuracy attributes of AEC, AECSA, EPRC80 and EPRC128. We compare the results from these designs with the software implementation of extended precision recursive summation. The objective of the experiments is to study the effect of dataset parameters such as condition number and exponent difference on the relative error.

6.2.1 Datasets

In order to test the numerical efficacy of the new designs– AEC, AECSA, EPRC80 and EPRC128– we generated different random datasets using uniform pseudo-random number generation functions in Matlab[40]. These datasets cover a wide range of values and exponents. Some of these datasets contain only positive numbers while some contain well distributed positive and negative numbers. Thus the condition number of datasets is also varied. We also use sparse matrices from the University of Florida sparse matrix collection[39] as test cases on real scientific data with variable input set sizes.

Generated datasets and sparse matrices can be divided categorized as well conditioned and ill conditioned datasets. The condition number(κ) for a dataset can be defined as the ratio of summation of absolute values and summation of input values as given by Equation 6.1. A dataset is said to be well-conditioned for which the condition number, κ, is close to 1.0. If all the values in a dataset have the same sign, κ = 1.0. If $\kappa \gg 1.0$, the dataset is said to be ill-conditioned and has good distribution of positive and negative values. Generally, if a dataset is well conditioned, the relative error encountered during summation is small even without accuracy preserving measures, while the relative error is large for ill-conditioned data. In ill-conditioned datasets, due to the presence of almost equal values with opposite sign, catastrophic cancellation occurs. Thus, using ill-conditioned datasets, the effect of catastrophic

cancellation on relative error can also be studied. We have used both well-conditioned and ill-conditioned datasets for our tests.

$$\kappa = \frac{\sum_{i=1}^{n} |a_i|}{|\sum_{i=1}^{n} a_i|} \tag{6.1}$$

In order to study the effect of shift during addition on the errors, we have used datasets with different exponent ranges. If dataset has values with wide range of exponents, the error due to shifting should be large. If the exponent difference in a dataset is small, error due to shifting is expected to be small. We have used datasets with known exponent ranges to measure the effect of shifting. The occurrence and magnitude of the error due to rounding operation cannot be predicted on the basis of the exponent range. The generated datasets consists of 1000000 entries and each is divided in sets of size 100 and 10000 values. The sparse matrices already have well defined rows thus check the designs for correct functionality with variable set sizes.

6.2.2 Experimental Setup

In order to measure the accuracy of results from our designs, we calculate the recursive summation of the sets in datasets in extended precision using GNU Multiple Precision Floating-Point Reliable Library (MPFR) with 2048 bit mantissa[38]. MPFR provides subroutines for arbitrary precision arithmetic on floating point numbers and is used to implement multi-precision floating point arithmetic in software. The precision can be chosen by the user.

In IEEE-754 double precision, the maximum permissible exponent is 1023 and the least exponent value is -1024. The maximum difference between exponents thus can be 2047. Thus, with 2048 bit mantissa, all the bits of the operand with smaller exponent are preserved and are not discarded during the shift operation. Also, rounding is not required because the results are not required to be truncated as sufficient precision is

available. Thus all the calculations in extended precision with 2048 bit mantissa can be considered infinitely precise with respect to the IEEE-754 double precision format.

For results from software, we first convert the double precision input values to extended precision values with 2048 bit mantissa. In recursive summation, where the sum (partial) of all the previous inputs is added to the current input value, all the intermediate operations are performed on values represented in this extended precision. The final results are converted to double precision format. These results are compared against the results from our designs. We report the number of incorrect bits with respect to the results from extended precision results. The relative error can be calculated using Equation 6.2.

$$\frac{|E_n|}{|S_n|} = \frac{|S_n| - S_n}{|S_n|} \tag{6.2}$$

The number of erroneous bits in the result can be calculated using Equation 6.3.

$$NumberofErrorneousBits = lg(2 \times \frac{|E_n|}{\epsilon.|S_n|}) \tag{6.3}$$

The intermediate calculations in 2048 bit precision mitigate the effect of shifting and rounding if the original inputs are in double precision as the shift amount cannot be more than 2047 bits and the partial results need not to be rounded. This approach suffers from loss of accuracy only once when the conversion is done from extended format to double precision format but the overall effect of this conversion is significantly less than multiple rounding operations and loss of bits due to shift when in double precision. We use the results from recursive summation in the 2048 bit extended precision as exact results for calculating the relative error.

In order to generate the results from our designs, we simulated the designs using Modelsim SE 6.6a. Text files containing the data is read by the top level module. This module then supplies the data to the designs. The output from the designs are stored in different files. The results from these files and those from software version of extended precision recursive summation are compared using Perl scripts.

6.2.3 Results

6.2.3.1 Random Datasets

In this section, we report the average number of erroneous bits for datasets generated using uniform random number generator. We report the condition number, exponent difference and error with respect to results from software expressed in terms of machine precision for each of the designs. The exponent range denotes the maximum possible exponent difference in the dataset. We can divide the datasets into three categories– $\kappa = 1.0$ and increasing exponent difference, $\kappa = \infty$ and increasing exponent range and increasing κ and fixed exponent range. It must be noted that the exponent range may vary during the summation of the dataset.

Table 6.4a show the average erroneous bits for set size of 100 and $\kappa = 1.0$ while Table 6.4b shows the average erroneous bits for set size of 10000. It can be observed that the error for set size of 100 is very small for all the designs. The effect of increasing the set size is evident on the results from the reduction circuit and AECSA. The reduction circuit suffers the most while the effect on AECSA is restricted to 3 bits. The results from AEC, EPRC80 and EPRC128 can be termed 100% accurate for both the set sizes as the error for set size of 100 is less than 1 bit for these designs.

Table 6.5a and Table 6.5b depict the number of erroneous bits for $\kappa = \infty$ and increasing exponent range. The erroneous bits have been calculated using absolute error E_n. These results show that the error increases as we increase the exponent range. Also, the effect of catastrophic cancellation is evident on results from the reduction circuit as it reports large erroneous bits for both set sizes. AEC and AECSA produce almost identical results and have the error restricted to 2 bits. EPRC128 have 100% accurate results while EPRC80 produces 100% accurate results in all but one case. This can be attributed to few large errors contributing to the average.

Table 6.4: Erroneous Bits for $\kappa = 1.0$ and Varying Exponent Difference

(a) Set Size=100

κ	**Exponent Range**	**Reduction Ckt.**	**AEC**	**AECSA**	**EPRC80**	**EPRC128**
1	2	0.9	0.0	0.1	0.0	0.0
1	4	0.8	0.0	0.0	0.0	0.0
1	8	1.4	0.0	0.3	0.0	0.0
1	16	1.8	0.6	0.7	0.1	0.1
1	32	1.7	0.0	1.0	0.0	0.0
1	64	1.9	0.6	0.8	0.3	0.3

(b) Set Size=10000

κ	**Exponent Range**	**Reduction Ckt.**	**AEC**	**AECSA**	**EPRC80**	**EPRC128**
1	2	14.2	0.0	2.1	0.0	0.0
1	4	14.5	0.0	2.8	0.0	0.0
1	8	16.96	0.0	2.1	0.0	0.0
1	16	17.61	0.0	3.0	0.0	0.0
1	32	18.87	0.0	3.2	0.0	0.0
1	64	18.51	0.0	3.2	0.0	0.0

Table 6.5: Erroneous Bits for $\kappa = \infty$ and Varying Exponent Difference

(a) Set Size=100

κ	**Exponent Range**	**Reduction Ckt.**	**AEC**	**AECSA**	**EPRC80**	**EPRC128**
∞	2	4.3	1.0	1.1	0.0	0.0
∞	4	4.8	1.1	1.1	0.0	0.0
∞	8	5.1	1.4	1.5	0.0	0.0
∞	16	5.9	1.4	1.6	0.0	0.0
∞	32	5.9	1.5	2.1	0.0	0.0
∞	64	8.3	1.8	2.3	0.0	0.0

(b) Set Size=10000

κ	**Exponent Range**	**Reduction Ckt.**	**AEC**	**AECSA**	**EPRC80**	**EPRC128**
∞	2	8.2	1.8	2.2	0.0	0.0
∞	4	9.2	2.8	3.2	0.0	0.0
∞	8	11.5	4.8	5.3	0.0	0.0
∞	16	15.5	5.7	5.9	0.0	0.0
∞	32	23.3	6.9	7.1	1.7	0.0
∞	64	26.2	9.7	9.7	0.0	0.0

Table 6.6a and 6.6b shows that if datasets have all the values with equal exponent, the increase in the condition number does not affect designs with accuracy improvement mechanisms much but the error for the reduction circuit increases as the condition number is increased. Please note that the condition number reported in these tables is the average condition number for the sets.

Table 6.6: Erroneous Bits for Varying κ and Exponent Range = 0

(a) Set Size=100

Exponent Range	κ	**Reduction Ckt.**	**AEC**	**AECSA**	**EPRC80**	**EPRC128**
0	10.2	1.9	0.3	0.6	0.0	0.0
0	99.6	3.4	0.4	0.6	0.0	0.0
0	651.3	4.6	0.7	0.8	0.1	0.1
0	1793.5	6.3	0.7	0.9	0.3	0.2
0	5896.2	8.5	1.0	1.0	0.8	0.7
0	11552.0	10.5	1.1	1.3	1.0	1.0

(b) Set Size=10000

Exponent Range	κ	**Reduction Ckt.**	**AEC**	**AECSA**	**EPRC80**	**EPRC128**
0	10.9	6.7	1.2	1.6	0.6	0.5
0	95.0	7.0	1.7	1.5	0.7	0.5
0	602.1	7.8	1.9	1.8	1.0	1.0
0	1796.5	9.3	1.9	2.1	1.2	1.1
0	5902.4	10.1	2.4	2.4	1.8	1.6
0	11603.4	10.5	2.8	3.1	1.8	1.7

Table 6.7a and Table 6.7b show that error reported by AEC, AECSA, EPRC80 and EPRC128 are significantly less that that from the reduction circuit even if we increase the exponent range. As such, the error reported by all the designs increases with the condition number. Also, as the set size increases, the number of erroneous bits also increase.

It can be observed from the results that in all the cases, the designs with accuracy improving mechanisms perform significantly better than the reduction circuit which does not employ any such mechanism. Also, the errors reported by AEC and AECSA are restricted to 4 bits or less in all the cases while EPRC80 and EPRC128 have 3 or less erroneous bits on average in all the cases. This clearly shows that these designs

Table 6.7: Erroneous Bits for Varying κ and Exponent Range = 32

(a) Set Size=100

Exponent Range	κ	Reduction Ckt.	AEC	AECSA	EPRC80	EPRC128
32	10.9	1.6	0.0	0.0	0.0	0.0
32	95.0	3.5	0.3	0.3	0.2	0.2
32	803.4	4.2	0.6	0.7	0.3	0.3
32	1612.5	7.7	0.7	0.9	0.4	0.4
32	5983.2	7.9	1.1	1.4	1.1	0.9
32	10972.8	8.3	1.6	2.5	1.4	1.4

(b) Set Size=10000

Exponent Range	κ	Reduction Ckt.	AEC	AECSA	EPRC80	EPRC128
32	10.2	10.6	2.8	2.8	2.7	2.7
32	95.8	10.3	3.2	3.2	3.2	3.2
32	801.2	11.7	2.5	3.0	2.3	2.2
32	1603.5	12.5	3.9	4.1	3.0	2.8
32	6002.7	13.2	3.7	4.1	3.1	2.9
32	10998.5	14.1	4.0	4.0	3.2	3.2

are not prone to the effects of cancellation and large shift amount. These designs offer significantly more accurate results.

Among AEC, AECSA, EPRC80 and EPRC128, AECSA produces most number of erroneous bits. This can be attributed to the large difference between the input and error term as the error compensation takes place in subsequent addition. EPRC80 and EPRC128 enjoy the advantage of having large number of extra bits and less number of bits are discarded during alignment operation. This also reduces the number of times rounding is required. Thus these designs produce the best results.

6.2.3.2 Sparse Matrices

In order to test the correctness of AEC, AECSA, EPRC80 and EPRC128 with variable length datasets and real scientific data, we simulated these designs for 10 different sparse matrices. The sparsity of these matrices does not affect the overall functionality of these designs. We calculate summation of each row for the sparse matrices. All the designs operate as expected and produce correct results for the sparse matrices.

In Table 6.8a, we report the properties of sparse matrices including dimensions, average number of non-zero entries per row (NZ). We present the condition number (for rows), κ, the average number of incorrect bits for reduction circuit, AEC, AECSA, EPRC80 and EPRC128 in Table 6.8b. On the basis of results for the sparse matrices, we can draw similar observations as drawn for randomly generated datasets. It can be seen that the relative error of the reduction circuit, AEC and AECSA, the error is high for matrices with large condition number of the rows. EPRC80 and EPRC128 provide more accurate results as compared to the other designs. If we compare the designs based on compensated summation, the results from AEC have lesser error than those from AECSA.

As such, the difference between the relative errors from the reduction circuit and the new designs is more than 50%. For the reduction circuit, the error increases with increasing condition number in all but one case. The anomaly can be attributed to the exponent difference.

Table 6.8: Relative Errors for Sparse Matrices

(a) Properties of Sparse Matrices

Matrix	Dimension	NZ	NZ/Row
psmigr_3	3140x3140	543162	172.9
cari	400x1200	152800	382.0
qc2534	2534x2534	463360	182.9
rat	3136x9408	268908	85.7
conf5.0-00l4x4-2200	3072x3072	119808	39.0
raefsky1	3242x3242	294276	90.8
heart	2339x2339	682797	291.9
qc354	324x324	26730	82.5
fidap	441x441	26831	60.8
fp	7548x7548	848553	112.4

(b) Number of Incorrect Bits for Sparse Matrices

Matrix	κ	**Exponent Range**	**Reduction Ckt.**	**AEC**	**AECSA**	**EPRC80**	**EPRC128**
psmigr_3	1	19	1.5	0.1	0.1	0	0
cari	1.4	14	1.3	0	0	0	0
qc2534	2.5	24	1.6	0.1	0.2	0.1	0.1
rat	3.5	12	1.7	0.3	0.5	0.2	0.2
conf5.0-00l4x4-2200	2.10E+01	17	2.8	1.1	1.3	0.9	0.7
raefsky1	2.20E+02	72	4.9	1.9	2.4	1.7	1.4
heart2	1.30E+03	36	9.4	2.3	2.9	2	1.9
qc354	2.50E+04	20	12.7	3.2	4.1	2.5	2.3
fidap002	2.80E+16	75	15.1	5.9	6.3	3.7	3.5
fp	3.50E+16	57	14.5	5.8	6.3	3.7	3.6

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The goal of this research was to study methods for improving the accuracy of summation and implement such methods for streaming set-wise floating point summation on FPGAs. To the best of our knowledge, this is the first effort to implement the compensated summation methods for set-wise floating point summation on FPGAs. In this dissertation, we studied methods to improve the accuracy of floating point summation. We presented a novel reduction circuit which is designed around a deeply pipelined floating point adder and dynamically accumulates the set-wise data. We implemented the designs for improving the accuracy of summation using the reduction circuit as the basic framework. We presented a custom floating point adder which along with the sum also output the floating point error. We use this adder in the two designs based on compensated summation namely AEC and AECSA in order to reduce the overall latency. We use different approaches in AEC and AECSA. We also presented two designs namely EPRC80 and EPRC128 where we use wider adder to perform all the intermediate additions in extended precision floating point format. We compared these designs with the original reduction circuit and found that these designs achieve significantly more accurate results at the expense of increased resource requirement. Barring AECSA, rest of the designs operate at a frequency which is comparable to the original reduction circuit. This demonstrates that better accuracy in floating point summation can be achieved without sacrificing the overall

performance.

Our experiments show that the condition number has significant impact on the relative error. The difference between the error for the designs with accuracy improving techniques and the reduction circuit is significant. Although AEC and AECSA are prone to the cancellation during summation, the effect is marginal when compared to the reduction circuit. EPRC80 and EPRC128 are less prone to this effect. This shows the effect of condition number on the error in floating point summation. Also, the error can be reduced using the compensated summation and extended precision.

The compensated summation methods are prone to the cancellation during floating point addition in such cases. AEC, which provides more accurate results with less relative error than AECSA, is also a viable option as the operating frequency is only slightly less than the original reduction circuit.

Our designs in which we used techniques to improve accuracy not only have significantly less relative error but also produce high number of exact results i.e. the number of erroneous bit is zero. We expected EPRC128 to give 100% exact results in all the test cases for both the sparse matrices and randomly generated datasets. This can be attributed to the presence of partial sums with opposite signs and the rounding which takes place when converting back to double precision format. This effect is not evident for randomly generated datasets.

Table 7.1 summarizes the designs implemented for improving accuracy and compares them with the reduction circuit. EPRC128 provides the best results amongst all the designs for all the test cases but requires almost three times the resources. Amongst the designs based on compensated summation, AEC performs better than AECSA contrary to the initial prediction. This is because the magnitude of error for compensation is very small in many cases and it remains unaccounted for. Also, the error compensation does not take place in every reduction as explained earlier. In such cases, the error is accumulated but partially accumulated errors have not proven

Table 7.1: Comparison of Designs

Design	Resources	Operating Frequency	Accuracy
AEC	Almost 3 times than that for reduction circuit	Slightly less	Relative errors are significantly less.
AECSA	Almost 4 times than that for reduction circuit	Significantly less	Errors comparable to AEC.
EPRC80	Almost 1.5 times than that for reduction circuit	Slightly less	Control logic is the same as original reduction circuit. Results are better than AEC and AECSA.
EPRC128	Almost 3 times than that for reduction circuit	Slightly less	Control logic is the same. Results are better than AEC, AECSA and EPRC80.

to be as effective as compensation in subsequent addition for AECSA.

Our experiments suggest that all the designs implemented achieve significantly better results than those from the reduction circuit. But for datasets having very large condition number, designs with extended precision– EPRC80 and EPRC128– perform better than the designs with compensated summation methods. It can be used in applications where it is not feasible to use extended precision floating point adder.

7.2 Future Work

In this dissertation, we considered two compensated summation methods for implementation on FPGAs. We realize there is a room for improvement. In the future, we would explore other compensated summation methods which are less prone to the effects of cancellation. The basic concern with compensated summation methods is the number of floating point operations and the overall latency of the network. Due to the increased latency, the control logic for error compensation becomes complex as seen in AECSA. We would target reducing the latency of the floating point adder without affecting its performance. Also, we plan to integrate these designs in the

system for sparse matrix vector multiplication.

Bibliography

[1] Nicholas J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput*, 14:783–799, 1993.

[2] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002.

[3] Ivo Babuska. Numerical stability in mathematical analysis. In *IFIP Congress (1)*, pages 11–23, 1968.

[4] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[5] Peter Linz. Accurate floating-point summation. *Commun. ACM*, 13(6):361–362, June 1970.

[6] Michael A. Malcolm. On accurate floating-point summation. *Commun. ACM*, 14(11):731–736, November 1971.

[7] Siegfried M. Rump, Takeshi Ogita, and ShinâĂŹichi Oishi. Accurate floating-point summation. Technical report, ABCD, 2005.

[8] Peter Kornerup, Vincent Lefevre, Nicolas Louvet, and Jean-Michel Muller. On the computation of correctly rounded sums. *IEEE Transactions on Computers*, 61(3):289–298, 2012.

[9] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18:305–363, 1996.

[10] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–145. IEEE Computer Society Press, 1991.

[11] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971/72.

[12] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[13] T. G. Robertazzi and S. C. Schwartz. Best "ordering" for floating-point addition. *ACM Trans. Math. Softw.*, 14(1):101–110, March 1988.

[14] Jack M. Wolfe. Reducing truncation errors by programming. *Commun. ACM*, 7(6):355–356, June 1964.

[15] James Gregory. A comparison of floating point summation methods. *Commun. ACM*, 15(9):838–, September 1972.

[16] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008.

[17] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40–, January 1965.

[18] W. Kahan and CALIFORNIA UNIV BERKELEY DEPT OF COMPUTER SCIENCES. *A Survey of Error Analysis*. Defense Technical Information Center, 1971.

[19] J.H. Wilkinson. Error analysis of floating-point computation. *Numerische Mathematik*, 2(1):319–340, 1960.

[20] James H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover Publications, Incorporated, 1994.

[21] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[22] Intel, Santa Clara, CA, USA. *IntelÂő 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*, March 2013.

[23] R. Hyde. *The Art of Assembly Language: Text.* ITPro collection. No Starch Press, Incorporated, 2003.

[24] Ling Zhuo, Gerald R. Morris, and Viktor K. Prasanna. Designing scalable fpga-based reduction circuits using pipelined floating-point cores. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3 - Volume 04*, IPDPS '05, pages 147.1–, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Ling Zhuo, Gerald R. Morris, and Viktor K. Prasanna. High-performance reduction circuits using deeply pipelined operators on fpgas. *IEEE Trans. Parallel Distrib. Syst.*, 18(10):1377–1392, 2007.

[26] Miaoqing Huang and David Andrews. Modular design of fully pipelined reduction circuits on fpgas. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2012.

[27] S. R. Vangal, Y. V. Hoskote, N. Y. Borkar, and A. Alvandpour. A 6.2-GFlops Floating-Point Multiply-Accumulator With Conditional Normalization. *IEEE Journal of Solid-State Circuits*, 41(10):2314–2323, October 2006.

[28] M. E. T. Gerards, J. Kuper, A. B. J. Kokkeler, and E. Molenkamp. Streaming reduction circuit. In *Proceedings of the 12th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools, Patras, Greece*, pages 287–292, Los Alamitos, August 2009. IEEE Computer Society.

[29] M. E. T. Gerards. Streaming reduction circuit for sparse matrix vector multiplication in fpgas. Master's thesis, University of Twente, August 2008.

[30] Michael deLorimier and André DeHon. Floating-point sparse matrix-vector multiply for fpgas. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, pages 75–85, New York, NY, USA, 2005. ACM.

[31] Nachiket Kapre. Optimistic parallelization of floating-point accumulation. In *In 18th Symposium on Computer Arithmetic*, pages 205–213. IEEE, 2007.

[32] Chuan He, Guan Qin, Mi Lu, and Wei Zhao. Group-alignment based accurate floating-point summation on fpgas. In *ERSA'06*, pages 136–142, 2006.

[33] Manouk V. Manoukian and George A. Constantinides. Accurate floating point arithmetic through hardware error-free transformations. In *Proceedings of the 7th*

international conference on Reconfigurable computing: architectures, tools and applications, ARC'11, pages 94–101, Berlin, Heidelberg, 2011. Springer-Verlag.

[34] Edin Kadric, Paul Gurniak, and Andre DeHon. Accurate parallel floating-point accumulation. In *Proceedings of the 2013 IEEE 21st Symposium on Computer Arithmetic*, ARITH '13, Washington, DC, USA, 2013. IEEE Computer Society.

[35] Yong-Kang Zhu and Wayne B. Hayes. Algorithm 908: Online exact summation of floating-point streams. *ACM Trans. Math. Softw.*, 37(3):37:1–37:13, September 2010.

[36] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[37] Krishna K. Nagar and Jason D. Bakos. A high-performance double precision accumulator. In *Proceedings of the The 2009 International Conference on Field-Programmable Technology*, FPT '09, 2009.

[38] The GNU MPFR Library. http://www.mpfr.org, Aug. 2013.

[39] Timothy A. Davis and Yifan Hu. The university of Florida sparse matrix collection. In *ACM Trans. Math. Softw.* 38, 1, Article 1 (December 2011).

[40] MATLAB version 7.10.0. Natick, Massachusetts: The MathWorks Inc., 2010.

[41] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. In *IEEE Design and Test of Computers*, 28(4):18–27, 2011