Delft University of Technology

Faculty of Electrical Engineering, Mathematics, and Computer Science
Department of Quantum & Computer Engineering

Enabling High Performance Posit Arithmetic Applications Using Hardware Acceleration

Thesis by:
Laurens van Dam

Advisors:
Prof. dr. H. P. Hofstee
Dr. ir. Z. Al-Ars

Committee:
Prof. dr. H. P. Hofstee (Chair)
Dr. ir. Z. Al-Ars
Dr. ir. M. Möller

Enabling High Performance Posit Arithmetic Applications Using Hardware Acceleration

THESIS

submitted in partial fulfillment of the requirements for the degrees of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

&

EMBEDDED SYSTEMS

by

LAURENS VAN DAM
born in SPIJKENISSE, THE NETHERLANDS

to be defended publicly on September 17, 2018 at 15:00.

Student number:	4203321	
Submission date:	September 10, 2018	
Responsible professors:	prof. dr. H. P. Hofstee	IBM Research, TU Delft
	dr. ir. Z. Al-Ars	TU Delft
Thesis committee:	prof. dr. H. P. Hofstee	IBM Research, TU Delft, jury chairman
	dr. ir. Z. Al-Ars	TU Delft, jury member
	dr. ir. M. Möller	TU Delft, jury member

CE-MS-2018-07
ISBN 978-94-6186-957-9
Quantum & Computer Engineering
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Abstract

The demand for higher precision arithmetic is increasing due to the rapid development of new computing paradigms. The novel posit number representation system, as introduced by John L. Gustafson, claims to be able to provide more accurate answers to mathematical problems with equal or less number of bits compared to the well-established IEEE 754 floating point standard.

In this work, the performance of the posit number format in terms of decimal accuracy is analyzed and compared to alternative number representations.

A framework for performing high-precision posit arithmetic in reconfigurable logic is presented. The supported arithmetic operations can be performed without rounding off intermediate results, minimizing the loss of decimal accuracy. The proposed posit arithmetic units achieve approximately 250 MPOPS for addition, 160 MPOPS for multiplication and 180 MPOPS for accumulation operations.
A hardware accelerator for performing Level 1 BLAS operations on (sparse) posit column vectors is presented. For the calculation of the vector dot product for an input vector length of 10^6 elements, a speedup of approximately $15000\times$ compared to software is achieved. The decimal accuracy is improved by one decimal of accuracy on average compared to posit emulation in software, and two additional decimals of accuracy are achieved compared to calculation using the IEEE 754 floating point format.

A study of the application of posit arithmetic in the field of bioinformatics is performed. The effect on decimal accuracy of the pair-HMM forward algorithm by replacing traditional floating point arithmetic with posit arithmetic is analyzed. It is shown that the maximum achievable decimal accuracy using posit arithmetic is higher compared to the IEEE floating point format for the same number of required bits.

The design of a hardware accelerator for the pair-HMM forward algorithm using posit arithmetic is proposed for two different interfaces: a streaming-based accelerator and an accelerator interfacing with Apache Arrow columnar data, both connected by the CAPI (SNAP) platform. Overall, the posit number format beats the IEEE floating point number format in terms of decimal accuracy, ranging from an improvement of 0.5 to 1 additional decimal of accuracy for the performed test cases. A throughput of 1.6 and 1 giga cell updates per second is measured for both accelerator implementations, respectively.

Preface

Before you lies the final product of my work of the last nine months, titled "*Enabling High Performance Posit Arithmetic Applications Using Hardware Acceleration*".

I could not have accomplished this work without the strong support from people I care about. First, my parents, who have always supported me with love and understanding. Secondly, my friends, who were there to support me during my endeavors (and to distract me once in a while).

My special thanks goes to Jinho Lee for the help with testing and debugging my hardware designs.

Thank you Matthias Möller for agreeing to be part of the graduation committee and providing me with useful research directions.

Lastly, I would like to thank my thesis supervisors, Peter Hofstee and Zaid Al-Ars, each of whom has provided patient guidance throughout this research process.

Thank you all for your unwavering support.

Laurens van Dam
Delft, The Netherlands
September 10, 2018

Contents

List of Figures

List of Tables

List of Acronyms

BID	Binary Integer Decimal
BLAS	Basic Linear Algebra Subprograms
CAPI	Coherent Accelerator Processor Interface
CUPS	Cell Updates Per Second
DPD	Densely Packed Decimal
FMA	Fused Multiply-Add
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
FTZ	Flush-To-Zero
GATK	Genome Analysis Tool Kit
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
HMM	Hidden Markov Model
INDEL	Insertion/Deletion
LSB	Least Significant Bit
LUT	Lookup Table
MAC	Multiply-Accumulate
MMIO	Memory Mapped Input/Output
NaN	Not-A-Number
PE	Processing Element

PHMM Pair Hidden Markov Model

POPS Posit Operations Per Second

PSL Power Service Layer

SA Systolic Array

SIMD Single Instruction, Multiple Data

SNP Single Nucleotide Polymorphism

SORN Sets Of Real Numbers

ULP Unit of Least Precision

WED Work Element Descriptor

Chapter 1

Introduction

1.1 Motivation

The demand for higher precision arithmetic units is increasing due to the rapid development of new computing paradigms. The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is the floating point implementation standard that the vast majority of contemporary computing systems have adopted. Although the first version of the standard was published in 1985, its main characteristics have to date not been changed, mainly for compatibility reasons. Because we now have more computing power at our disposal, the ability to process larger amounts of data becomes possible. In addition to this, we impose large precision requirements, either while accumulating big portions of data or when there is the desire to have a high-precision calculation result while imposing strict boundaries on bandwidth and the available processing power. This is often the case when limited computing resources are available, for example in low-power applications such as in embedded systems.

Over the course of the past few decades, the IEEE floating point representation has become the de facto standard in modern computing systems. Although it has proven to be a popular standard for representing floating point numbers, multiple shortcomings have been identified that have not been treated as major issue in the past, mainly because the standard was simply "good enough" for the conditions and requirements imposed at that time. A selection of the shortcomings of the IEEE floating point standard are listed below [1].

- Different computers using the same IEEE floating point format are not required to produce the same result. If a computation result does not fully fit into the chosen number representation, the number will be rounded to the nearest representable value. Hidden guard digits were introduced to improve the accuracy of those rounded off answers. However, hardware designers are not obliged to implement these hidden guard digits. Therefore, inconsistent results can occur across different computing platforms.

- Non-conformance of the law of Associativity and Distributivity: due to the fact that rounding is performed on individual operands of a specific calculation, the basic mathematical laws of associativity, commutativity and distributivity do not hold. This means that, for instance, the sum of two numbers in floating point representation does not necessarily match the exact sum of those two numbers if they were to be represented as exact numbers.

- A portion of the available number of bit patterns is "wasted" on exceptions. An example of an exception is the Not-A-Number (NaN) value, which is used to represent an undefined or unrepresentable number. This is often the result of having illegal arguments as an input to a mathematical function. For instance, dividing by zero leads to a NaN. The IEEE 754 floating point standard defines different types of NaN representations. This is discussed in detail in Section 2.2.1.

The aforementioned discussion regarding the shortcomings of the widely adopted IEEE 754 standard was what led to the idea of developing an alternative number representation system that would serve as a worthy successor. The recently introduced *posit* number representation system was introduced by John L. Gustafson and claims to be able to provide more accurate answers to mathematical problems with an equal or smaller number of bits for some applications [2]. The posit data type is discussed in Section 2.1.3. Furthermore, the discussion on the shortcomings of the floating point standard and how the posit data type fixes these shortcomings is described. A key feature of this format is the *quire*, a scratchpad that can be used to perform fused operations, such as a multiply-accumulate operation, without rounding intermediate values and therefore reducing loss of precision in the final result.

The differences in semantics of the posit number format compared to the traditional IEEE 754 floating point number format lead to the question whether the posit number format might be a suitable replacement for applications that rely on precision. Furthermore, applications with smaller accuracy requirements could also benefit from a more accurate number representation system as a smaller number of bits could be used compared to floating point numbers, while preserving results with tolerable precision.

1.2 Thesis Aim & Contributions

The main goal of this thesis is to explore a selection of potential applications for the novel posit number representation system. The potential advantages of the posit representation are exploited through the design of multiple hardware accelerator designs. These accelerators are built upon the most recent Coherent Accelerator Processor Interface (CAPI) in order to establish a high-speed connection between a host and accelerator. Furthermore, the accelerators presented in this work will be able to interface with data represented in the Apache Arrow columnar in-memory format.

Research Questions

The main research questions for this thesis are formulated as follows.

1. Study the suitability of the posit number representation for replacing traditional IEEE 754 floating point numbers in terms of accuracy and performance.

2. Does the posit number format deliver better, more accurate results compared to the IEEE 754 floating point standard for the following applications:

    a) Pair Hidden Markov Model (pair-HMM) for pairwise alignment of DNA sequence reads

    b) Level 1 BLAS Vector Operations (addition/subtraction, multiplication and dot product)

3. Can the computational performance of the posit implementation of the applications as mentioned in (2) be improved through acceleration using reconfigurable logic?

Research Methodology

The procedure to address the above mentioned research questions is described as follows. First, a theoretical analysis is performed on the definition of the posit number representation format. As the IEEE 754 floating point format is seen as the most popular alternative to the posit number format, a detailed analysis is made regarding the characteristics of both number formats. After analyzing the theoretical characteristics of both formats, a more practical exploration is performed in the context of the pair-HMM pairwise alignment algorithm. The algorithm is evaluated for both number formats and compared in terms of decimal accuracy of the calculation results. For the implementation of posit arithmetic in reconfigurable logic, a framework is designed that enables posit arithmetic in hardware with a focus on the preservation of decimal accuracy. Using this framework, an implementation of an accelerator for the pair-HMM algorithm using posit arithmetic is designed, as well as an accelerator for performing Level 1 Basic Linear Algebra Subprograms (BLAS) operations on column vectors consisting of posit numbers. Both hardware implementations are evaluated based on performance and the achieved decimal accuracy. Finally, a conclusion is made based on the obtained results.

Contributions

The contributions described in this thesis can be summarized as follows.

- A feasibility study to explore potential improvement in accuracy of computation results when replacing traditional floating point numbers with numbers represented in the posit number format.

- Design, implementation and evaluation of a novel framework for performing posit arithmetic operations in hardware while minimizing loss in decimal accuracy caused

by intermediate calculations.
`https://github.com/lvandam/posit_arith_hdl`

- Design, implementation and evaluation of a vector arithmetic accelerator for posit vectors interfacing with the Apache Arrow columnar in-memory format through the CAPI SNAP platform.
  `https://github.com/lvandam/posit_blas_hdl`

- Design, implementation and evaluation of a hardware accelerator for the pair-HMM forward algorithm using posit arithmetic for two different interfaces:
  - A streaming-based implementation interfacing through the CAPI platform.
    `https://github.com/lvandam/pairhmm_posit_hdl_stream`
  - Interfacing with data structures represented in the Apache Arrow columnar in-memory format through the CAPI SNAP framework.
    `https://github.com/lvandam/pairhmm_posit_hdl_arrow`

1.3 Thesis Outline

This thesis is structured as follows. Chapter 2 discusses background information related to the proposed unum arithmetic framework. Next, a comprehensive analysis of the posit number format compared to the IEEE 754 floating point standard is performed. The chapter is concluded with an overview of existing methods for controlling the desired numerical precision in software applications during the compilation time of a program. The feasibility of applying posit arithmetic in the field of bioinformatics is discussed in Chapter 3. In particular, the Pair Hidden Markov Model and its computational implementations are discussed. A detailed theoretical and empirical analysis is performed in order to determine the potential improvement in terms of calculation precision. Chapter 4 discusses the design, implementation and evaluation of a novel framework for performing posit arithmetic in reconfigurable logic. The framework consists of posit arithmetic units that can be used for performing calculations that are optimized with respect to decimal accuracy. The design and implementation of a hardware accelerator for performing posit vector arithmetic operations are discussed in Chapter 5. Subsequently, the design, implementation and evaluation of an accelerator for the pair-HMM forward algorithm using posit arithmetic in discussed in Chapter 6. An overall conclusion on the analysis, design and implementation efforts presented in this thesis, including recommendations based on the performed analyses, is given in Chapter 7. Additionally, a recap on the research questions initially proposed in the introduction of this thesis is given.

Chapter 2

Background

In this chapter we discuss background information related to the analyses performed and the implementations presented in this work. First, a description of the unum arithmetic framework, consisting of the proposals of unum type I, II and III (posit), is discussed. Next, a comparison between features of the IEEE 754 floating point standard and the novel posit number format is performed. Building upon the background theory on the posit number format, the recent efforts in hardware implementations of posit arithmetic are then discussed. We will also explore alternative number representation systems. In the concluding sections of this chapter we will cover background information on compiler optimizations for controlling desired floating point calculation accuracy, the Apache Arrow columnar in-memory format and a definition of the measure of decimal accuracy.

2.1 The Unum Arithmetic Framework

The shortcomings of the IEEE 754 floating point number standard as discussed in Section 1.1 have led to an alternative number format that could resolve those issues. The universal number, abbreviated as *unum* [1], is introduced by John L. Gustafson and is seen as one of the most promising alternatives to the IEEE standard that has been the standard for decades. The unum number format has evolved over the past few years, dividing the unum into three different iterations or types: type I, II and III.

2.1.1 Type I Unum

The main feature of a unum value compared to a regular IEEE 754 number is its ability to represent either an exact number or an open interval of 1 Unit of Least Precision (ULP) wide. The reason for wishing to incorporate this feature in the unum number scheme is that a computation is often not able to provide a numerically exact answer due to limitations in the number representation it uses. Therefore, when a number is unrepresentable in a certain degree of precision, it is rounded off to the nearest representable number. A unum is able to indicate that the value is accurate within a range of 1 ULP through the so-called *ubit*, which is a 1-bit field within the unum number representation. In this way,

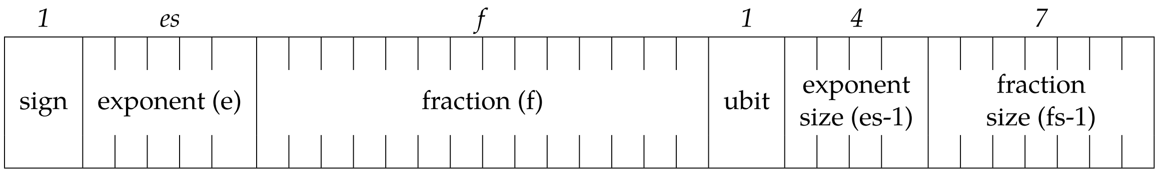


Figure 2.1: Schematic overview of the bit fields of a unum type I. The field widths are indicated above each field.

ubit $= 0$ indicates that the unum corresponds to an exact number while ubit $= 1$ indicates that the unum corresponds to an interval between exact unums.

The unum format borrows most of the components of the IEEE 754 floating point scheme, such as the *exponent* and *fraction* (or mantissa) fields. The distinctive feature of the type I unum, however, is the fact that the widths of those fields are variable. One could choose to either represent a large number by assigning more bits to the exponent field, or one could opt for more decimal precision by having more fraction bits. The *exponent size* and *fraction size* fields are added to the unum scheme in order to annotate the widths of the exponent and fraction fields, biased upward by 1 as there is always at least one exponent and fraction bit.

In order to cover all the bit field widths of the exponent field as defined in the IEEE 754 standard, having 4 bits to indicate the exponent size $(\text{es} - 1)$ is sufficient [1]. Similarly, having 7 bits to indicate the fraction size $(\text{fs} - 1)$ is sufficient to cover the different IEEE floats. The complete format of unum type I is schematically depicted in Fig. 2.1.

2.1.2 Type II Unum & SORNs

Motivation & Shortcomings of Unum Type I

The proposal of the type I unum proved to be a promising new number representation system, but was not free of any drawbacks. Implementing type I unum arithmetic in hardware is particularly challenging [3]. For example, the fact that unums can have a variable number of total bits and exponent bits leads to the requirement having variable storage sizes available. As implementing dynamic storage in hardware is challenging, one would need to unpack unums to a fixed storage size using a specific scheme [1]. Furthermore, the *ubit* field in a unum always has to be determined before the remaining fields of a unum can be unpacked. This introduces more complexity to a hardware implementation. For example, the ubit introduces more comparisons that have to be performed compared to a hardware implementation for floats. Another drawback of type I unums is that certain values can be represented in different ways, i.e. with different combinations of the number of total bits and exponent bits. Therefore, a number could be represented by using fewer exponent bits compared to another possible representation for the same number that would fill up a bigger portion of the fraction field. The second version of unum, type II, was proposed to resolve some of the shortcomings described above.

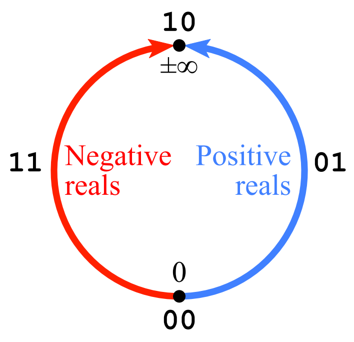


Figure 2.2: Visual representation of the projective real number line of type II unums in the case of a 2-bit unum type II [3].

Description

One of the distinctive features of the type II unum is its different mapping of unum values onto the projective real line. This mapping consists of a line with negative and positive reals, where the two ends meet at a single point, representing either plus or minus infinity. A visualization of how unums are mapped to the real line is depicted in Fig. 2.2. As can be seen, the point where a 2's complement number changes from positive to negative is the same point as where the positive reals wrap into the negative reals. This point also represents the value $\pm\infty$.

Another component the type II specification introduces is the Sets Of Real Numbers (SORN). The SORN is a bit string that represents whether or not a region or subset is present (1) or absent (0) in a given range. A SORN is therefore able to define a specific interval using subsets of the projective real numbers. The subsets of the projective reals depicted in Fig. 2.2 are $\pm\infty$, $(-\infty, 0)$, 0 and $(0, \infty)$. If one were to use a SORN to encode a unum with an interval of $(-\infty, 0]$, the SORN would be 0110 (thereby indicating that the subsets $(\infty, 0)$ and 0 are part of the interval).

As noted by the author, the SORN can be used for operations that would normally result in indeterminate forms with single numbers. Expressions that would usually result in an indeterminate form such as zero divided by zero or infinity minus infinity are valid expressions when using SORNs because they produce valid SORNs. A SORN, for example, is able to represent the range of all numbers (by including all subsets). Analogously, for a division by zero, one can take the limit to 0 and end up with either or minus infinity. A SORN is then able to represent the value $\pm\infty$.

In order to perform arithmetic operations on SORNs it is necessary to perform table lookups to determine the outcome of any SORN operation on two or more operands. An example of such an operation (addition) is depicted in Fig. 2.3. Since these lookups need to be frequently performed, it is imperative to make this fast and efficient. When implementing unums in hardware these Lookup Tables (LUTs) could, for instance, be implemented in ROMs.

An interesting property that arises from the way in which the projective reals are mapped (as described above) is that there are geometrical analogies about the horizon-

+	▮ ○ ▯ ○	▯ ● ▯ ○	▯ ○ ▮ ○	▯ ○ ▯ ●
▮ ○ ▯ ○	▮ ● ▮ ●	▮ ○ ▯ ○	▮ ○ ▯ ○	▮ ○ ▯ ○
▯ ● ▯ ○	▮ ○ ▯ ○	▯ ● ▯ ○	▯ ● ▯ ○	▯ ● ▮ ●
▯ ○ ▮ ○	▮ ○ ▯ ○	▯ ● ▯ ○	▯ ○ ▮ ○	▯ ○ ▯ ●
▯ ○ ▯ ●	▮ ○ ▯ ○	▯ ● ▮ ●	▯ ○ ▯ ●	▯ ○ ▯ ●

Figure 2.3: Two-input SORN addition table. Red and blue represent the negative and positive values, while a circle represents an open interval and a rectangle an exact value [3].

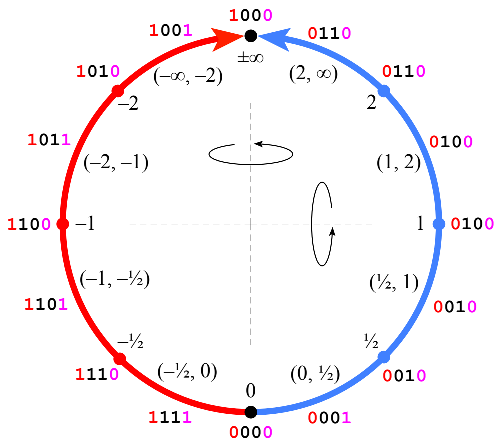


Figure 2.4: The mapping to projective reals of a 4-bit type II unum, showing the symmetries along the horizontal (negation) and vertical (reciprocal) axes [3].

tal and vertical axes of the unum representation. Take for instance the four-bit unums depicted in Fig. 2.4. When flipping about the horizontal axis, the values are negated. Analogously, flipping about the vertical axis results in the reciprocal value. Like type I unums, the *ubit* field classifies a unum as representing an open interval (ubit $= 1$) between adjacent exact points (ubit $= 0$).

Critique

One can imagine that the proposal of a new, alternative number representation system that directly competes with a settled number format (the IEEE 754 floating point standard) sprouts discussions between proponents of both number formats. Prof. W. Kahan, one of the lead architects on the initial version of the IEEE 754 floating point standard [4], has publicly expressed his critique on the proposed unum type II number format that includes the concept or SORNs. His critique led to a major discussion between proponents of both the IEEE 754 and unum number representation systems. We will now review a summary of Kahan's opinions and arguments as described in a document published in 2015 [5] and presented at ARITH 23 in 2016 [6].

The following arguments are directed towards unum type I and II (SORNs), for which some have been addressed in the type III (*posit*) number format as discussed in the next section.

- The *algebraic integrity* is violated with SORNs, as different expressions or the same rational function could produce different SORN results.
- Unum computation involves a too high cost:
  - The latency compared to floats is increased because of the additional unpacking pipeline stages for variable-width unums.
  - Fetching unums costs at least one extra indirect address reference compared to floats due to the variable fraction and exponent sizes.
- The area cost for SORN arithmetic can be significantly large: depending on the number of available collections of SORNs, chip area might increase significantly. Arbitrary SORN collections would require $\mathcal{O}(2^{3N})$ area for SORNs each represented by 2^N bits wide words. Schemes with pairs of N-bit pointers would need $\mathcal{O}(N \times 2^{2N})$, but this is slower. This might be good for low-precision interval arithmetic. However, low-precision interval arithmetic might not be used at all.
- The debugging of SORNs and unum arithmetic might be difficult:
  - Lengthy computations with SORNs would produce too wide intervals which are difficult to diagnose and/or resolve.
  - The IEEE 754 flags are absent in SORNs and unums: wide SORNS or unums caused by underflow or overflow would be difficult to diagnose and/or resolve as there are no pointers to the location of the first occurrence of an exception. Furthermore, no Not-A-Number (NaN) value exists for unums, which might make it difficult to discover invalid operations.

2.1.3 Type III Unum (Posit)

Motivation

Although type II unums were praised for their mathematical properties, one of the biggest drawbacks is the reliance on lookup tables in hardware implementations. For an n-bit type II unum there is a worst-case LUT size of 2^{2n} per 2-argument function scenario [2]. As discussed in Section 1.1, one of the most desired features of a number format is to have support for fused operations. However, type II unums prove to be unsuitable for use in fused operations due to the fact that any unum can have a variable number of (exponent) bits. This makes it difficult to store unums with different configurations in one accumulator. A successor to type II unums was therefore proposed. Type III unums, also known as *posits*, would take the general ideas and advantages of the type I and II unums to a point where hardware implementations supporting posit arithmetic would be very similar to the existing logic used for IEEE 754 floating point arithmetic [2].

Description

Compared to types I and II unums, the scheme of posits changed drastically. The posit format consists of the *sign*, *regime*, *exponent* and *fraction* fields.

Sign The sign bit is 0 for positive numbers and 1 for negative numbers, in which case the 2's complement form for the remaining part of the posit has to be taken before extracting the remaining fields.

Regime The regime field is used for calculating a scale factor k. This scale factor is determined by the number of repetitions of the leading bit in the regime field. If the regime field consists of leading 1's, this number determines the value k after subtracting 1. For a number of leading 0's, the number is negated. For example, if a 4-bit regime field is set to 1110, then $k = 2$. Analogously, $\mathrm{regime} = 0001$ would indicate $k = -3$. Therefore, a bit that is opposite to the other leading bits terminates the regime field and marks the first position of the next field (exponent).

Exponent Similar to the type I and II unums, the exponent field has a variable width and determines the scaling factor 2^{es} of the value represented by a posit. Recall that type I unums have a dedicated field to indicate the width of the exponent field. Posits do not require this field, as the number of exponent bits (es) is known in advance. Furthermore, the first bit of the exponent field is located directly after the regime field (which is dynamically terminated by a bit opposite to its leading bits, as mentioned above).

Fraction The remaining bits that have not been occupied by the sign, regime and exponent occupy the fraction field. The format for the fraction bits is the same as for IEEE floating point numbers (also sometimes termed *significand* or *mantissa*). The value of the fraction field is normalized, i.e. the represented decimal value starts with $0.\mathrm{xxx}$.

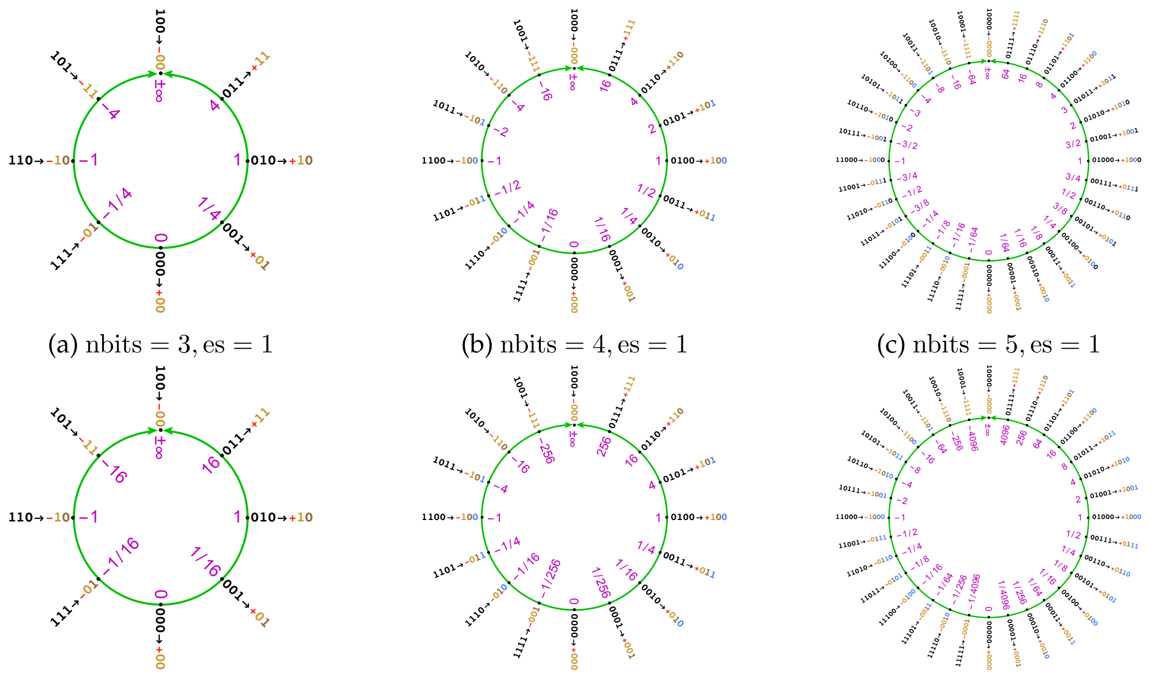
The decimal value of a posit is then calculated as

$$(-1)^{sign} \times useed^{k} \times 2^{e} \times (1 + f) \tag{2.1}$$

where:

$useed$	the scaling factor determined by the number of exponent bits (es), equal to $2^{2^{\mathrm{es}}}$
k	the value of the *regime* field
e	the value of the *exponent* field
f	the value of the *fraction* field

Fig. 2.5 shows the representable values for a selection of posit configurations. The bit strings around the outside of the rings can be treated as 2's complement integers. As can be seen, the point where the integers transition from positive to negative values is the same point as where the represented numbers transition from positive to negative as well. In much the same way as the characteristics of unum type II (Section 2.1.2), flipping a posit number around the vertical axis yields its negative value. For 0, $\pm\infty$ and powers of two flipping across the horizontal axis yields the reciprocal.



(a) nbits = 3, es = 1 (b) nbits = 4, es = 1 (c) nbits = 5, es = 1

(d) nbits = 3, es = 2 (e) nbits = 4, es = 2 (f) nbits = 5, es = 2

Figure 2.5: Representable values for combinations of $\mathrm{nbits} = 3, 4, 5$ and $\mathrm{es} = 1, 2$ [7].

Fused Operations

When performing any calculation on one or multiple operands, rounding is performed in order to fit the final result back into a specific format. Intermediate rounding of results can therefore be a source of accuracy loss. This is especially the case when calculation results are being used as input operands for next calculations. Multiple approaches exist in order to defer the rounding of results until the very last calculation, such as the Fused Multiply-Add (FMA) operation (refer to Section 2.5.3). The proposal of the posit number representation system describes the concept of the *quire*, which can be seen as a fixed-size scratchpad register that can be used for different operations. The size of this scratchpad register is wide enough to obviate the need of having to round intermediate answers. The calculation of the required quire width for a specific calculation is discussed in Appendix B. Although the concept of scratchpad registers that can be used for fused operations is not novel, the concept of the quire yields a slightly different approach. While existing scratchpad registers are not controllable on the user-level, a quire unit would be accessible by the user. Hence, the user could instantiate a quire and provide a reference to this quire in any calculation he would like to perform without rounding of calculation results.

Valids

The specification for the posit number system also describes the notion of the *valid* mode, representing a number range that can be used for interval arithmetic. In its simplest form, a valid consists of two posits indicating the start and end of a bound. Valids are therefore useful for computations that require bounds on their results. Any loss of accuracy is tracked, since the inaccuracy bound will increase whenever any uncertainty is added to the overall computation. Another advantage of applying valids would be to verify the correct numerical behavior of a program, thereby keeping track of how much inaccuracy a program introduces in its intermediate computations [7]. The resulting bound could therefore serve as an indication to perform calculations with higher precision in order to decrease the error bounds for precision-critical applications.

2.2 Feature Comparison of IEEE Float and Posit

As the proposal for the posit number format advertises itself as an alternative for the IEEE 754 floating point standard, direct comparisons are often made between both number formats. In this section we will discuss the major differences between both formats on an architectural level. Furthermore, we take a look at past discussions between opponents and advocates of the novel number format.

2.2.1 Not-A-Number (NaN)

The IEEE 754 standard has defined that an exceptional number (such as NaN or $\pm\infty$) should be represented by having all exponent bits set to one [4]. The remaining sequence of bits (fraction bits) then determines the exception cases: if the bit sequence is set to zero, the complete bit pattern represents $\pm\infty$, depending on the sign bit. A non-zero sequence represents NaN (as discussed in Chapter 1). The first bit of this bit string indicates the type of NaN, which can be either *quiet NaN* (propagating through arithmetic operations without an exception being raised) or *signaling NaN* (signaling an invalid operation exception). The remaining fraction bits are to be used as a payload, which could for example be used to indicate where in the program the NaN has occurred. However, this is often not used in practice as there is no user-friendly software-defined interface for programmers to control the payload (mantissa) field of a NaN floating point value. As the payload field of a NaN value can yield any value, there are different implementations for several programming languages that each have their own way of making use of the "unused" payload field of NaN values. This method is otherwise known as *NaN boxing* and is heavily used in JavaScript engines, among others [8]. The idea of "hacking" numbers via the principles of NaN boxing draws negative feedback. John L. Gustafson for example notes that this "breaks just about every rule there is about creating maintainable standards" [1].

In the case of a 32-bit single-precision IEEE 754 floating point number, there are 23 fraction bits available. Since the value zero is used to represent $\pm\infty$, this means that there are $2^{23} - 2 = 8\,388\,606$ different bit patterns to represent NaN. This is approximately 0.2%

of the total number of representations possible for a 32-bit single-precision number. One would argue that it is a waste of bit patterns to dedicate this amount to solely represent an invalid number.

As discussed in Section 2.1.3, the posit number format employs a different view on the use of NaN values. Instead of encoding different types of NaN values (such as quiet or signaling NaN), the posit arithmetic framework has no bit representations for NaN values at all [7]. The rationale behind this is that cases where a program returns a NaN value during or after calculation should not occur. Therefore, an interrupt should be asserted whenever a situation arises where an invalid value is the result of a computational routine. This interrupt can be handled by the program to, for instance, run a backup routine that acts as a ”workaround” to still be able to generate a valid outcome.

There are multiple advantages of not having to represent NaN values in a number representation system. For example, the arithmetic hardware becomes less complex as less checks have to be performed to identify a bit string as either a number or a special value. Furthermore, the number of bit patterns that would have been used to represent NaN (or any other special value) can be used to represent a bigger range of numbers. As discussed, the 32-bit single precision IEEE 754 format has approximately 0.2% of all bit patterns reserved for NaN values. For posits, this portion can actually be used to represent more numbers.

2.2.2 Negative Zero and $\pm\infty$

As discussed in the analysis in Section 2.1.3, positive values change to negative values at the same point as for 2's complement integers. This is contrary to the IEEE 754 floating point standard which also contains a value to represent the value negative zero. This value is not representable in posits. Analogously, posits have a single representation for the value $\pm\infty$ (similar to the wraparound at $\pm\infty$ shown in Fig. 2.2) whereas IEEE floats have separate values for ∞ and $-\infty$.

2.2.3 Critique

As discussed in Section 2.1.2, William Kahan, principal architect of the IEEE 754 floating point standard, has expressed a considerable amount of critique on the unum number format. The posit (type III unum) format addresses multiple points of criticism from Kahan, including the removal of interval arithmetic using the concept of SORNs (refer to Section 2.1.2). Furthermore, as opposed to the unum type II proposal, unum bit lengths are not variable any more and instead calculations are performed for a pre-selected fixed posit configuration. Kahan's suggested alternative solution to accomplishing more reliable arithmetic compared to the current state-of-the-art is by investing in software support for IEEE 754 diagnostics such as the NaN flags that point to sites where the first arithmetic exception has occurred. As discussed earlier in this chapter, the signaling NaN values intended for diagnostics purposes are often either not supported or used for other purposes.

2.3 Posit Arithmetic Hardware Implementations

Since the initial introduction of the posit number format in mid-2017, the amount of research and development related to this novel number representation has been increasing rapidly. Starting from the introduction of the type I unum, a range of different hardware implementations for arithmetic units have been proposed, including hardware arithmetic implementations for the type III unum (posit). Recent developments in the design of posit arithmetic units have resulted in a range of different approaches to implementing posit arithmetic operations such as additions and multiplications in hardware.

The design of the posit matrix-multiply unit presented by Chen et al. [9] proves to be a high-performance implementation of posit arithmetic, achieving approximately 10 GFlops on an IBM POWER8 platform featuring the CAPI 1.0 interface. The Coherent Accelerator Processor Interface (CAPI) 1.0 interface is used to feed input data from the host to the accelerator. In this work, a quire register (refer to Section 2.1.3) is implemented that enables vector dot products to be performed without intermediate rounding. For the Xilinx Virtex-7 VX690 FPGA, the reported effective performance is 16 GPOPS (Giga Posit Operations Per Second) when streaming in posit elements and performing a multiply-accumulate operation. This implementation is limited to 32-bit posit numbers with $\mathrm{es} = 2$.

As discussed in Section 2.1.3, one of the key features of the posit number format is the ability to adjust the total number of bits used to represent a number, along with the amount of bits used to represent the number exponent (similar to the IEEE floating point format). The ability to deal with posit numbers of variable-width fields becomes challenging when designing a hardware implementation of posit arithmetic due to the fixed, static nature of hardware. Although earlier work has focused on designing posit arithmetic units for reconfigurable logic with a configurable number of (exponent) bits using a synthesis parameter, being able to provide accurate computation results for all input combinations still proves to be challenging. For instance, the arithmetic hardware generator with configurable number of posit bits as proposed by [10] is not synthesizeable for any posit configuration with zero exponent bits. Furthermore, the posit arithmetic units presented in this work do not apply a rounding scheme. Instead, fraction values are simply truncated when normalizing to an N-bit posit word. Moreover, the presented posit arithmetic unit implementations are not pipelined, placing an entire calculation in a single combinational path, and as such are not directly useable in high-speed designs. For the proposed posit multiplier with the `posit<32,2>` configuration, a latency of approximately $16\,\mathrm{ns}$ is reported for the Xilinx Virtex-6 Field-Programmable Gate Array (FPGA), resulting in a operating frequency of $62.5\,\mathrm{MHz}$.

2.4 Alternative High-Precision Number Formats

With the IEEE 754 floating point standard being the most popular number representation system, alternative number formats that can be used in software are often based on this standard. The reason for this is the fact that integrated Floating-Point Units (FPUs), which are designed specifically for performing operations on floating point numbers, exist since the era of the x87 subset of the x86 architecture, providing an instruction set related to performing floating point operations in hardware [11].

Number formats that are not based on the IEEE 754 floating point format often rely on software emulation due to the lack of dedicated hardware support. Software emulation of number representation formats often appears to be significantly slower than hardware floating point operations, as we will see for the emulation of a posit dot product calculation discussed in Section 5.3.

In this section, we discuss a selection of alternative number formats, based on the IEEE 754 standard, that are designed to perform calculations with more decimals of accuracy.

2.4.1 Decimal Floating Point

The decimal floating-point format is part of the 2008 version of the IEEE 754 specification [12]. The format differs from other floating point representations in the sense that the decimal format is able to emulate decimal rounding. Hence, floating point numbers are rounded exactly on decimals, which is often wanted in financial applications. Furthermore, the number fields are not normalized. This makes it possible for multiple representations to exist for the same number.

The significand of a decimal floating point number can be represented by two different methods, being Binary Integer Decimal (BID) or Densely Packed Decimal (DPD). However, both representations result in the same range of representable values. For BID, the significand is represented by a positive integer in binary representation. In the DPD representation, the significand is stored as decimal digits. Hence, in order to represent a single decimal digit, 4 bits are required per digit. The Densely Packed Decimal (DPD) [13] encoding is used in order to encode the digits after the most significant digit in order to save bandwidth.

Although the decimal floating point number format proves to be useful for applications critical for exact decimal calculations, there are a number of disadvantages for using this concept.

Hardware Support Several checks have to be performed before performing calculations. For example, it is determined which significand representation is used. Furthermore, not all possible combinations inside a decimal number are supported. For example, the BID significand representation does not support significand values larger than $10^{34} - 1$. Any values above this limit are treated as zero. These checks need to be performed in hardware and thus can be a source for performance issues or an increase in area usage.

Rounding Errors Extreme rounding errors may occur. As significands do not have to be normalized, a large error can occur when adding values with different exponents. This is caused by the fact that the smallest number is shifted in order to match both exponents. After addition, the result is rounded up to a fixed number of digits. As the bits truncated in the final result are weighted by a factor of 10^e, the decimal error can increase significantly. An example 7-digit calculation is depicted below, illustrating the possibility of having large rounding errors.

$$\begin{array}{llll} & 1.460147 \times 10^6 + & 2.194512 \times 10^2 & \\ = & 1.460147 \times 10^6 + & 0.0002194512 \times 10^6 & \textit{(shift)} \\ = & 1.460366\textcolor{red}{4512} \times 10^6 & & \\ = & 1.460366 \times 10^6 & & \textit{(round)} \end{array}$$

2.4.2 Boost Multiprecision

The *Boost* C++ libraries [14], a widely used set of C++ extensions, provides the *Multiprecision* library that is designed to provide arithmetic types with a high number of decimals of accuracy. The `cpp_dec_float` type, for instance, claims to provide at least 100 decimals of precision. Internally, `cpp_dec_float` numbers are represented in radix-10, similar to the decimal floating point format discussed before. Similar to IEEE 754 floating point numbers, both infinity and NaN values are supported. As reported in the Boost documentation, multiple guard digits are implemented in order to reduce error when values are being truncated.

2.5 Compiler Optimization

Although the semantics of the employed number representation system are of great importance to calculation accuracy, it is often possible to adjust the desired level of precision of computation results through available options that can be set at the compiler level. In this section we will explore a selection of compiler options that can be used to control the desired level of floating point calculation precision.

2.5.1 Optimization Level

It is often possible to choose the desired floating point semantics at compile-time of a program by setting corresponding compiler flags. This way, the user is able to control the level of granularity of floating point calculations to a certain degree. For the majority of the compilers available, the amount of optimization that is performed is controlled through the *optimization level* flag. For an optimization level of 0 (-O0), optimization transformations on a program are minimized. Each increasing level of optimization might improve

Optimization	Description
`no-math-errno`	Single-instruction math operations do not set the *errno* (Error Number) field
`unsafe-math-optimizations`	Arithmetic input operands and results are assumed valid without validation
`finite-math-only`	No checks are performed for NaN or $\pm\infty$ numbers, allowing finite numbers only
`no-rounding-math`	Assume default rounding behavior when performing optimizations
`no-signaling-nans`	Reduced number of NaN values that would generate user-visible traps
`excess-precision=fast`	Compute with wider precision whenever this would result in a faster program

Table 2.1: Effect of the GCC `fast` Optimization Level on floating point calculation semantics for the GCC compiler [15].

program performance, while potentially giving in on accuracy of floating point calculations. For the GNU Compiler Collection (GCC), the optimization levels $0-3$ do not affect floating point arithmetic behavior. However, the `fast` optimization level does enable several optimizations regarding floating point semantics. The optimizations that are in effect for this mode are depicted in Table 2.1.

2.5.2 Subnormal Numbers

Arithmetic underflow can occur when a calculation result becomes smaller than the actual representable value in a certain number representation. Depending on the distribution of the representable numbers by a number representation system, the *underflow gap* between the smallest positive representable number and the smallest negative representable number is several orders of magnitude larger than the interval of one ULP in the regular, non-overflow case. Multiple approaches to dealing with underflowing values exist. Historically, underflow values were treated as zero. This feature, also known as Flush-To-Zero (FTZ), is still a configurable option in common compilers through a FTZ flag. In particular, asserting the FTZ flag flushes underflowing floating point computation results to zero [16]. The main advantage of setting the FTZ flag is the potential performance improvement. As more values are treated as zero, fewer computations have to be performed. However, this does induce a potential loss of precision, as information is being discarded. Therefore, the flush-to-zero option gives the user control over a trade-off between program performance and the desired level of accuracy.

An alternative to the traditional flush-to-zero behavior, the concept of *subnormal numbers*, was introduced in the IEEE 754 floating point specification and fills in the underflow gap described above [17]. Subnormal numbers are values smaller than the smallest "normal" representable value in a number representation system. For the IEEE 754 floating point standard, a subnormal value is indicated by setting the exponent field to the smallest representable value. Although the mantissa of a normal floating point value is represented by a hidden 1 bit, subnormal numbers are represented by a hidden 0 bit and therefore allow for smaller numbers to be represented. The introduction of subnormal numbers enables *gradual underflow*: instead of underflowing to zero, the nearest subnormal value is used [18]. Although the process of gradual underflow induces a loss of precision, the severity of the precision loss is minimized since a flush-to-zero is avoided.

2.5.3 Multiply-Accumulate

Multiply-Accumulate (MAC) is an instruction designed to compute the product of two input numbers and add the result to an *accumulated* value. Hence, the MAC operation performs the following calculation:

$$z \leftarrow z + x \times y \tag{2.2}$$

The MAC instruction rounds the result of the product $(x \times y)$ to a specific number of significant digits, followed by an addition to the accumulated value z, after which rounding to a specific number of significant digits is performed again. This procedure is alternatively called *double rounding*. It is trivial to see the disadvantage of having the double rounding scheme in place for the MAC operation. For every MAC operation, two roundings are performed, which can in turn cause a significant error.

As opposed to the double rounding scheme for the MAC operation, the FMA operation performs rounding only once: the complete multiply-accumulation $x \times y + z$ is calculated and then rounded to a specific number of significant digits. The FMA operation was initially introduced as part of the RISC instruction set of the IBM POWER1 processor [19]. Subsequently, the FMA instruction has found its way into nearly all contemporary processors. The FMA operation has been part of the IEEE 754 floating-point standard since 2008 [12].

2.6 Measure of Decimal Accuracy

In order to express the error between a value and an exact reference value, a wide range of measures exist. For example, the absolute error, calculated as the absolute difference between a value and its reference, is a popular error metric. Building upon this definition, the relative error gets rid of the order of magnitude of both numbers by dividing the absolute error by the exact value. This measure, however, is particularly unsuitable for quantifying the number of digits that are correct between a computed and exact value, as only a ratio is given. Furthermore, the relative error is not informative for cases when the computed value has a different sign compared to the reference value as the absolute value is taken.

The previously described drawbacks of the commonly used measures for error lead to the proposal of the measure of *decimal accuracy* [7]. Suppose we have the true value X and the calculated value $\tilde{X}$, which for example can be the result of a calculation in float or posit mode. We can define the *decimal error* as the ratio between an exact and computed value:

$$\text{decimal error} = \left| \log_{10}\left(\frac{\tilde{X}}{X}\right) \right| \tag{2.3}$$

(a) Table

	Field 1	**Field 2**	**Field 3**
Row 1	1	2	3
Row 2	4	5	6
Row 3	7	8	9

(b) Regular Buffer

	1
Row 1	2
	3
	4
Row 2	5
	6
	7
Row 3	8
	9

(c) Arrow Buffer

	1
Field 1	4
	7
	2
Field 2	5
	8
	3
Field 3	6
	9

Table 2.2: Example table (a) represented in a traditional memory buffer (b) and an Apache Arrow columnar memory buffer (c).

The *decimal accuracy* is then defined as a measure of the number of decimals of accuracy and is equal to the log base 10 of the inverse of the decimal error [7]:

$$\text{decimal accuracy} = -\log_{10}\left|\log_{10}\left(\frac{\tilde{X}}{X}\right)\right| \tag{2.4}$$

The presented measure of decimal accuracy is able to express how many decimals in a computed value are accurate. For example, take a computed value of 1.001 and a reference value of 1. The decimal accuracy is then equal to approximately 3.362, indicating that approximately 3 decimals (including non-fractional decimals) are accurate.

2.7 Apache Arrow

The Apache Arrow [20] platform is a cross-language development platform designed for representing in-memory data. Software libraries are available for a wide range of programming languages, among others C, C++, Java and Python. The memory layout of Apache Arrow is designed to enable zero deserialization overhead when transferring data among different platforms (such as the big data platform Apache Spark). As opposed to a traditional memory buffer, Arrow buffers are organized with *columnar* data locality as is illustrated in Table 2.2. Data stored in a table-like structure is stored per column instead of per row. This brings an advantage when requesting data for a single column, as the rows inside each column are stored consecutively, avoiding the need of requesting a column for each row index. For the representation of data structures in Apache Arrow, a *schema* is defined. A schema can contain multiple *array*s. These arrays can be compared with a table-like structure: each array represents a separate table column. Each column is characterized by a number of parameters. The column *field* describes these parameters, and includes the name of the column as well as the logical type that is stored in the column. A column can be nested, i.e. it is able to hold children, which is particularly useful when constructing structures or lists.

Chapter 3

Applications in Bioinformatics

In this chapter, we explore the feasibility of implementing posit arithmetic in the field of bioinformatics. In particular, we explore the feasibility of applying posit arithmetic in the evaluation of the pair-HMM model. By simulating a set of pair-HMM likelihood computations in a software-based environment that is able to emulate the posit scheme we obtain a first impression of the performance of the posit number format in this application. We analyze the results of the empirical analysis in order to determine whether implementing posit arithmetic does indeed prove to be beneficial to the overall accuracy of pair-HMM computations. We use the measure of decimal accuracy as defined in Section 2.6 to compare the accuracy of numbers with a higher-precision reference calculation.

First, we discuss background information in the field of bioinformatics with a focus on the pair-HMM model. Next, we discuss past work on hardware acceleration of the pair-HMM algorithm. The feasibility of applying posit arithmetic in the pair-HMM forward algorithm is then explored by means of a theoretical as well as an empirical analysis of the performance of the posit number format compared to calculations performed using the traditional IEEE 754 floating point format.

As posits can be configured with variable number of total and exponent bits, we use the notation `posit<nbits,es>` for a posit with `nbits` total bits and `es` exponent bits.

3.1 Pair Hidden Markov Model (Pair-HMM)

A Hidden Markov Model (HMM) is a discrete Markov chain that is augmented with the concept of *hidden states*. The observation of a hidden state is non-deterministic: it is a probabilistic function of the state itself [21]. Consider a system with N states $S_1, S_2, \ldots, S_N$, and let the state transition probability from state i to j be a_{ij} with $a_{ij} > 0$, $\sum_{j=1}^{N} a_{ij} = 1$. This process has observable states, i.e. one is able to observe the current state and the order of states prior to the current state. That is, at each time step one is able to observe the current set of states.

The concept of Markov chains can be extended with the notion of hidden states: the stochastic process is extended with a second, unobservable (hidden) stochastic process.

x	G	T	A	T	G	A	-	-
y	-	-	A	T	G	A	T	A
z	I_x	I_x	M	M	M	M	I_y	I_y

Table 3.1: Example sequence observations x and y and their underlying sequence of hidden states z for the pair-HMM model illustrated in Fig. 3.1.

An example of a simple HMM is the observation of a coin toss result where the toss performance itself is not observable. Hence, a sequence of coin tossing experiments results in a set of outcome observations of heads and tails [22]. The previously described concept of HMMs can again be extended by having multiple observations instead of only a single observation. The resulting Pair Hidden Markov Model (PHMM) can be used for the generation of probability distributions for sequences of *pairs* of observations. This type of HMM is particularly useful for finding alignments between sequences, for example in DNA analysis when matching DNA reads with a specific haplotype sequence [23].

An example of a pair-HMM model is depicted in Fig. 3.1. This model consists of three states (I_x, I_y, M). States I_x and I_y are able to insert an (unaligned) symbol in sequence x and y respectively, while state M is able to insert an aligned symbol pair (x_i, y_i) in both sequences x and y. In this example, a direct transition from state I_x to I_y and vice versa is not possible.

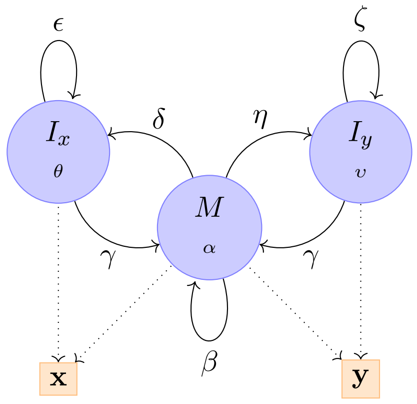


Figure 3.1: A pair-HMM with 3 states. Symbols can be inserted into sequences x or y. δ and η are the probabilities of emitting symbol x and y respectively, while α denotes the probability of emitting two aligned symbols into sequences $\mathbf{x}$ and $\mathbf{y}$. The remaining symbols denote the state transition probabilities.

Let the hidden state sequence for this pair-HMM model be denoted by z. A one-to-one relationship exists between z and the alignment of the two sequence observations x and y. This can be illustrated as follows. Consider the two observed sequences x and y and the underlying sequence of hidden states z depicted in Table 3.1. In this example, symbol pairs (x_3, y_1), (x_4, y_2), (x_5, y_3) and (x_6, y_4) have been emitted by hidden state A, indicating an alignment of these symbols. Based on this example, in order to find the best alignment between two sequence observations x and y we need to maximize the alignment probability resulting from the most optimal sequence of hidden states y.

It is often meaningful to determine whether two sequences are related, and not necessarily perfectly aligned. By summing over the possible state sequences we obtain the joint observation probability as [23]:

$$P(\mathbf{x}, \mathbf{y}|t, \epsilon, \pi) = \sum_{\mathbf{y}} P(\mathbf{x}, \mathbf{y}, \mathbf{z}|t, \epsilon, \pi) \tag{3.1}$$

where:

$\mathbf{x}$, $\mathbf{y}$	the two observed sequences
$\mathbf{z}$	the sequence of hidden states
t	the transition probabilities
ϵ	the emission probabilities
π	the initial state probabilities

The aforementioned pair-HMM model is especially suitable in the field of genome analytics, where the pair-HMM model can be used to align a read sequence against a set of candidate haplotypes. This process is used in order to assign genotypes to potentially variant regions. Given a set of candidate haplotypes, each haplotype needs to be evaluated in order to determine if a haplotype is indeed matches a specific read sequence. Therefore, the pair-HMM model can be used to align each individual read base pair against each candidate haplotype. Along with the read data sequence, metadata such as the base read and Insertion/Deletion (INDEL) quality scores (refer to Section 3.2) of each read determines the emission and transmission probabilities used in the pair-HMM model. The final output of the pair-HMM model then yields a likelihood of observing a specific read given a haplotype [24].

3.1.1 Pair-HMM Forward Algorithm

The pair-HMM model as described in Section 3.1 provides a method to compute the alignment probability between two observed sequences. The traditional method of evaluating the aforementioned pair-HMM model is through exploration of all possible states. The computational complexity of this *adhoc* method is equal to $\mathcal{O}(nm^n)$ [25], where n denotes the read sequence length and m the length of the haplotype sequence to compare with. As can be seen, the computational complexity rapidly increases for long input sequences as the number of possible state sequences increases. One of the possibilities to calculate the maximum alignment probability more efficiently is through dynamic programming. The pair-HMM *forward algorithm* is able to compute the maximum alignment probability of two sequences in a recursive manner. The pair-HMM forward algorithm is defined as follows. Refer to Section 3.1 for the pair-HMM model used in the following definition.

Define the following recursion variables:

$$\begin{aligned} M(i,j) &= \alpha_{i,j}\left[\beta_i M(i-1,j-1) + \gamma_i(I_x(i-1,j-1) + I_y(i-1,j-1))\right] \\ I_x(i,j) &= \theta_i\left[\delta_i M(i-1,j) + \epsilon_i I_x(i-1,j)\right] \\ I_y(i,j) &= \upsilon_j\left[\eta_i M(i,j-1) + \zeta_i I_y(i,j-1)\right] \end{aligned} \tag{3.2}$$

where:

$M(i,j)$	the combined probability of all alignments up to symbol (i,j) that end in state M
$I_x(i,j)$	the combined probability of all alignments up to symbol (i,j) that end in state I_x
$I_y(i,j)$	the combined probability of all alignments up to symbol (i,j) that end in state I_y
$\alpha_{i,j}$	the probability of emitting two aligned symbols into sequences $\mathbf{x}$ and $\mathbf{y}$
θ_i, υ_j	the probabilities of emitting symbol x_i and y_i respectively
$\beta, \gamma, \delta, \epsilon, \eta, \zeta$	the state transition probabilities

with initial conditions:

$$\begin{array}{rrr} M(0,0) = 1 & I_x(0,0) = 0 & I_y(0,0) = 0 \\ M(i,-1) = 0 & I_x(i,-1) = 0 & I_y(i,-1) = 0 \\ M(-1,j) = 0 & I_x(-1,j) = 0 & I_y(-1,j) = 0 \end{array} \tag{3.3}$$

Then

$$P(\mathbf{x},\mathbf{y}|t,\epsilon,\pi) = M(n,m) + I_x(n,m) + I_y(n,m) \tag{3.4}$$

where:

$P(\mathbf{x},\mathbf{y}\|t,\epsilon,\pi)$	the likelihood of sequences $\mathbf{x}$ and $\mathbf{y}$ being related
n	the length of the $\mathbf{x}$ sequence
m	the length of the $\mathbf{y}$ sequence

The computational complexity of the pair-HMM forward algorithm is $\mathcal{O}(nm)$ as this algorithm only scales linearly with both input sequence lengths n and m. This is significantly more efficient compared to the $\mathcal{O}(nm^n)$ complexity of the adhoc state exploration method described in Section 3.1. Due to the reduced complexity of the pair-HMM forward algorithm compared to the regular pair-HMM model, this algorithm is suitable for scientific computing. The pseudo code of an implementation of the pair-HMM forward algorithm is described in Appendix A.

Preventing Underflow

The pair-HMM forward algorithm as described in this section provides a time-efficient method for computing the alignment likelihood between an input read sequence and a reference haplotype. The forward algorithm is therefore often applied in genome analysis toolkits such as the Genome Analysis Tool Kit (GATK) [26]. Note that for small emission and transition probabilities, computation results of the recursion variables as defined in Eq. (3.2) can become very small since these calculations consist of multiple products of these emission and transmission probabilities. A possible consequence of computing with relatively small values is the risk of *underflow*: a computation result becomes too small to be representable in a specific number format. A common solution to prevent underflow during computation of the recursion variables of the pair-HMM forward algorithm is the *scaling* of the input values through the use of a chosen initial constant. The initial condition $M(0,0) = 1$ as defined in Eq. (3.3) can be set to a higher value (e.g. a power of 2) in order to scale all intermediate calculations by a fixed constant, preventing underflow as intermediate results are again representable in a specific number format [27].

3.1.2 Pair-HMM Hardware Acceleration

The pair-HMM model is a widely used model for pairwise alignment tasks that are often performed in the field of genome analysis, as discussed in Section 3.1. An example of such analysis toolkit is the GATK, as discussed in Section 3.2. While the pair-HMM model has proven to be a computationally well-suited algorithm, the performance can decrease significantly when considering large amounts of input data. The pair-HMM forward algorithm as discussed in Section 3.1.1 serves as an appropriate target for parallelization as the algorithm accomodates a form of intra-task parallelism through the anti-diagonal recurrence pattern of the pair-HMM forward algorithm.

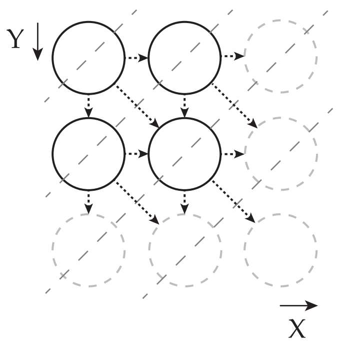


Figure 3.2: Anti-diagonal recurrence. The anti-diagonal dashed lines indicate computations that can be calculated in parallel. Dotted lines indicate a data dependency.

An illustration of the anti-diagonal recurrence pattern in the pair-HMM forward algorithm is depicted in Fig. 3.2. In this figure, each circle represents a base pair comparison task. The dotted arrows indicate a data dependency between base comparison tasks. As can be seen, one is able to exploit the parallelism among cells on the same anti-diagonal, as there are no computation dependencies among these cells. Apart from the intra-task parallelism the forward algorithm has a form of inter-task parallelism on the data level in the sense that every task where two bases are compared using the recurrence variables as described in Section 3.1.1 is independent of base comparison tasks that are performed for other bases.

Although pair-HMMs have been implemented for different applications for over a decade, research in accelerating the pair-HMM model through (reconfigurable) hardware is an ongoing effort. Accelerated implementations of the pair-HMM algorithm have been presented for Single Instruction, Multiple Data (SIMD) platforms such as Graphics Processing Units (GPUs) [28][29]. Furthermore, recent efforts in accelerating the pair-HMM forward algorithm show effective implementations using reconfigurable hardware such as Field-Programmable Gate Arrays (FPGAs) [30][31][32]. The majority of past efforts of implementing the pair-HMM forward algorithm in reconfigurable hardware exploit the intra-task parallelism, making use of the anti-diagonal recurrence pattern of the forward

	Pass			
	1		**2**	
↓ Y	1,1	2,1	3,1	4,1
	1,2	2,2	3,2	4,2
	1,3	2,3	3,3	4,3
	1,4	2,4	3,4	4,4
			X →	

Figure 3.3: Mapping of 2 processing elements (highlighted as white and gray) to base comparison tasks for a read and haplotype sequence length of 4 base pairs. As the number of PEs is smaller than the input sequence length, two passes are required.

	PE		
	1	**2**	**3**
↓ *t*	1,1		
	1,2	2,1	
	1,3	2,2	3,1
	1,1	2,3	3,2
	1,2	2,1	3,3
	1,3	2,2	3,1
		2,3	3,2
			3,3

Figure 3.4: Mapping to three processing elements (PEs) of two batches (highlighted white and gray) of a read and haplotype sequence of 3 base pairs each. One clock cycle is represented by each row.

algorithm as depicted in Fig. 3.2. This form of parallelism is most often implemented by an architecture based on a Systolic Array (SA). A SA consists of a network of coupled Processing Elements (PEs) that are able to work independently on an identical computation process. For the application of the pair-HMM forward algorithm, each PE computes one element of the match, insertion and deletion matrices of the pair-HMM model as described in Section 3.1.1.

For the example shown in Fig. 3.2, one could implement one PE for every horizontal cell (in the X-direction) in order to calculate the matrix elements of one anti-diagonal per cycle update. In case a SA consists of less PEs than the largest input sequence length, multiple passes are required in order to compute all matrix elements of the forward algorithm. This is illustrated in Fig. 3.3.

Although the SA structure proves to be an effective way of parallelizing pair-HMM forward algorithm computations in hardware, a common drawback of this structure is the potential underutilization of resources, for example when only a part of the SA is working on calculating the alignment probability between two pairs while the remaining PEs could have been working on calculations for the pairs of the next alignment task already. The pair-HMM SA design as proposed by Peltenburg et al. [31] is optimized for SA utilization by being able to work on multiple pairs in parallel. Hence, PEs are able to work on the next batch while other PEs might still be working on the previous batch. This is illustrated in Fig. 3.4, where the next batch is scheduled for PEs that are done calculating matrix elements of the previous batch.

3.2 GATK

The Genome Analysis Tool Kit (GATK) [26] is a widely used toolkit for genome analysis, mainly containing tools for genotyping and variant discovery processes. The GATK analysis pipeline roughly consists of the following steps [33]:

1. **Pre-Processing**: The raw input sequence data is cleaned up based on a set of rules. Furthermore, the sequence data is aligned to a reference genome. This step consists of the following processes:

   a) *Map to Reference*: Individual read pairs in the input sequence data is mapped to a reference genome.

   b) *Mark Duplicates*: Before the variant discovery, any read pairs that originate from duplicate DNA fragments are tagged in order to be ignored in the variant calling step. These duplicates can arise during any of the preparation steps where DNA samples are prepared for reading.

   c) *Base Quality Score Recalibration*: Confidence scores that are given by the sequencer for every DNA sample read, often represented by the Phred scale (discussed in Section 3.3.2), could contain a systematic bias. This systematic bias needs to be corrected in order to obtain reliable quality scores that can be used during the variant calling process.

2. **Variant Discovery**: Variations in one or multiple reads are identified compared to a reference genome, generating so-called *variant calls*.

   a) *Per-Sample Variant Calling*: On a per-sample basis, mutations in the input genome are detected (called). These mutations can be either a Single Nucleotide Polymorphisms (SNPs), being a single base being different compared to a reference genome, or an INDEL where a base is inserted or deleted in the input genome compared to the reference.

   b) *Joint Genotyping*: The variant calls produced in the previous step are collected and passed into a joint genotyping tool. In this process, SNPs and INDELs are collected and prepared for filtering. The result of this process is a set of genotypes for samples of interest.

   c) *Variant Quality Score Recalibration*: Similar to the Base Score Recalibration process in the Pre-Processing step, the variant callset produced in the previous steps are recalibrated in order to obtain more reliable quality scores that have been produced by the variant caller.

3. **Filtering/Annotation**: By combining known information about variations in the DNA that map to a specific disorder, filtering and annotation can be performed in order to obtain better readable results during downstream analysis. This information can also be combined with related metadata, resulting in additional information compared to the raw pre-processed genome data.

In the GATK, the *HaplotypeCaller* tool is used for the per-sample variant calling process (step 2a). Regions of the input genome sequence where one or multiple variants might be present are defined. Next, the reference haplotypes are determined. These reference haplotypes might be a match to the region that potentially contains variant(s). For every candidate haplotype, the pair-HMM algorithm as discussed in Section 3.1 is performed in order to perform pairwise alignment of the haplotypes to the input genome. The algorithm then returns a likelihood for each candidate haplotype. Candidate haplotypes can be converted to a likelihood for every potential variant in the read genome. Finally, the most likely genotype can then be assigned to the variant sample. The pair-HMM process in the HaplotypeCaller is seen as one of the performance bottlenecks in the GATK pipeline due to its computational complexity. Therefore, the pair-HMM algorithm is often targeted by accelerators.

3.3 Pair-HMM in Float and Posit Representation

The decimal accuracy of the calculation results produced by evaluating the pair-HMM model, as discussed in Section 3.1, is seen as an important factor in the process of DNA variant calling. More accurate model evaluations could result in more accurate predictions of a potentially variant site. The pair-HMM model therefore serves as one of the potential applications where posit arithmetic could potentially improve overall decimal accuracy. In this section, both a quantitative and an empirical analysis on the feasibility of applying posit arithmetic in this application is performed.

3.3.1 Quantitative Analysis

We perform a theoretical analysis in order to evaluate the behavior of the posit number format compared to the traditional IEEE 754 float format. More specifically, we will focus on the analysis of the application of the posit number format in the pair-HMM pairwise alignment algorithm (as discussed in Section 3.1).

Define the following variables:

n	the total number of bits in a posit configuration
es	the number of exponent bits in a posit configuration
k	the decimal value represented by the regime field

We define the minimum representable number posmin_k for a given regime value k as

$$\mathrm{posmin}_k = (2^{2^{\mathrm{es}}})^{k-1} \tag{3.5}$$

Furthermore, let fbits denote the number of fraction bits available in an n-bit posit number, which is equal to n subtracted by the number of bits used to represent the sign, regime and exponent:

$$\mathrm{fbits} = n - \mathrm{kbits} - \mathrm{es} - 1 \tag{3.6}$$

where kbits denotes the number of bits required to represent a regime k (including terminating bit) and is defined, according to the encoding specification of the regime field [2], as

$$\mathrm{kbits} = \begin{cases} k+2 & k \geq 0 \\ k+1 & \text{otherwise} \end{cases} \tag{3.7}$$

Then, we can define the relative change in magnitude Δ of a posit number as the log base 10 of the ratio between a posit value and its next representable value, where the fraction field is increased by one Least Significant Bit (LSB), given the total number of bits n, exponent bits es and regime k:

$$\begin{aligned}
\Delta_{\mathrm{n,es,e,k}} &= 10\log_{10}\left(\frac{\mathrm{value}_{f=1}}{\mathrm{value}_{f=0}}\right) \\
\Delta_{\mathrm{n,es,e,k}} &= 10\log_{10}\left(\frac{(2^{2^{es}})^k \cdot 2^e \cdot (1+2^{-\mathrm{fbits}})}{(2^{2^{es}})^k \cdot 2^e \cdot (1+0)}\right) \\
\Delta_{\mathrm{n,es,k}} &= 10\log_{10}(1+2^{-\mathrm{fbits}})
\end{aligned} \tag{3.8}$$

As can be deducted from the combination of Eq. (3.6), Eq. (3.7) and Eq. (3.8), the relative change in magnitude of a posit number is dependent on its regime k. It is therefore imperative to observe the posit number spectrum at specific boundary conditions. Therefore, we observe the following:

Observation 1 *Let p be a number representing an emission or transmission probability used in the pair-HMM forward algorithm, where $0 \leq p \leq 1$. Then, the regime value k of a posit representing p satisfies $k < 0$.*

As described in Observation 1, any posit number p representing a probability score in the pair-HMM application is within the range $0 \leq p \leq 1$, which implies the regime value k is negative. Hence, we focus our analysis on situations for negative regime values. We compare the relative change in magnitude Δ between posit values (for regime $k < 0$) and the IEEE 754 `float` number format in the same value range of $0 \leq p \leq 1$.

The relative change in magnitude of a 32-bit IEEE 754 floating point number can be defined as:

$$\begin{aligned}
\Delta_{\mathrm{float}} &= 10\log_{10}\left(1+2^{-\mathrm{fbits}}\right) \\
&= 10\log_{10}\left(1+2^{-23}\right) \approx 5.1772 \times 10^{-7}
\end{aligned} \tag{3.9}$$

Note that this value is constant: the number of fraction bits in the `float` format is fixed and equal to 27 [4].

In Fig. 3.5, the relative change in magnitude for the `posit<32,2>` and `posit<32,3>` number based on Eq. (3.8) is compared against the (constant) relative change in magnitude for the `float` format for different values of the *scale*. The scale represents the factor 2^k used in the conversion of a posit and IEEE 754 floating point format.

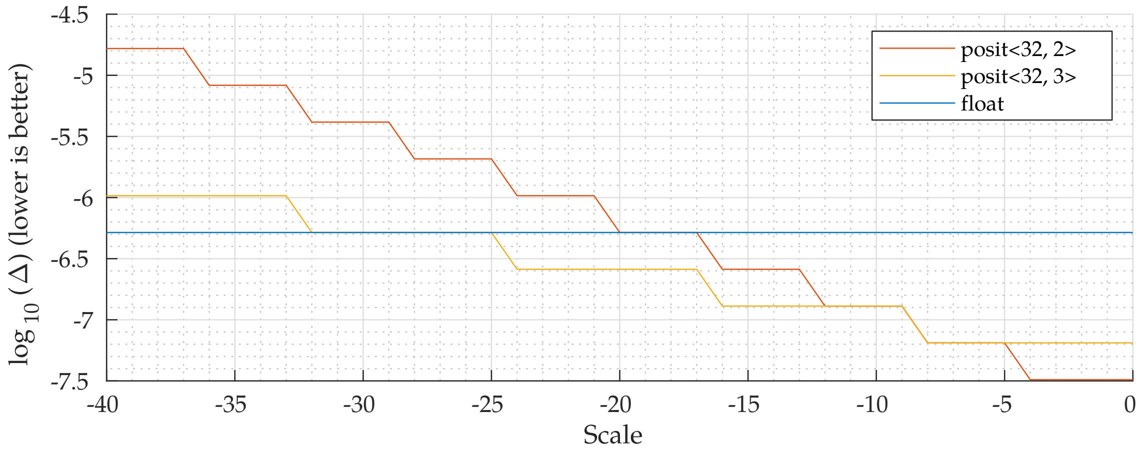


Figure 3.5: Relative change in magnitude for `posit<32,2>`, `posit<32,3>` and `float`.

For a posit value, the scale is calculated based on the regime and the exponent of the posit value. Since the regime value accounts for a factor of $(2^{2^{\text{es}}})^{\text{regime}}$ (where *es* denotes the configurable size of the exponent field) and the exponent value adds a factor of 2^{exponent}, combining these factors results in the following scale factor [34]:

$$\text{scale} = (2^{2^{\text{es}}})^{\text{regime}} \times 2^{\text{exponent}} = 2^{2^{\text{es}} \times \text{regime} + \text{exponent}} \tag{3.10}$$

For a IEEE 754 floating point value, the scale is equal to the factor $2^{\text{exponent}-127}$ and thus only depends on the exponent.

As can be concluded from Fig. 3.5, the relative change in magnitude for a posit number decreases for higher values of the scale. This can be explained by the fact that for values of the scale closer to 0, less bits are required to represent the regime k (refer to Eq. (3.7)), in which case more fraction bits are available to represent a wider range of fractions. Although a smaller change in magnitude for values of the regime k close to 0 is beneficial for minimizing decimal accuracy due to rounding errors, the minimum representable value increases (Eq. (3.5)). The relative change of magnitude of the `posit<32,2>` format becomes lower than the relative change in magnitude of `float` for scale values greater than or equal to -20. The relative change in magnitude for a `posit<32,3>` number is lower compared to the `float` format for $\text{scale} \geq -32$.

Summary

The analysis performed in this section focused on the measure of *relative change in magnitude*. For this measure, we have performed a comparison between the posit and IEEE 754 floating point number format. A small relative change in magnitude for an increment of one LSB in a number fraction is considered beneficial for reducing errors in decimal accuracy, as rounding errors are less penalized when contributing to an overall result. At the same time, the minimum representable number is desired to be as small as possible since lower probabilities (in the case of the pair-HMM forward algorithm) can then be represented. For the posit number format, both measures are controlled by the regime value and the exponent. For the representation of probabilities, the combined scale is negative in order to represent values between 0 and 1. However, the relative change in magnitude depends on the overall scale value as can be seen in Fig. 3.5. From this we can conclude that values exist that would induce less rounding error when representing the value in the `posit<32,2>` or `posit<32,3>` formats compared to the IEEE 754 `float` format, depending on the value of the regime and exponent.

3.3.2 Phred Scale

The *Phred scale* is a widely used scale for representing probabilities and confidence scores. In particular, the Genome Analysis Tool Kit (GATK) (as discussed in Section 3.2) uses this scale for representing a wide variety of probabilities and scores. Moreover, the read quality of DNA sequencing data, which is often represented by the Phred scale, is used heavily in the pair-HMM model for pairwise alignment.

Background

DNA reads that are performed by a DNA sequencer assigns individual scores to each base call, called the *base quality score*. These scores indicate the level of confidence for a specific base read. These scores are often represented in the Phred scale. Similar to the base quality scores, the *variant quality score* estimates the level of confidence of a *variant call*. Variant calling is a process where variants are being identified between a reference genome and a given sample. In this case, the Phred-scaled variant quality score determines the level of confidence that the variant has indeed been correctly detected.

The Phred quality score is based on a logarithmic scale, and is calculated based on the error probability. Let E be the error probability ($0 \leq E \leq 1$) that is to be represented in the Phred scale. The Phred quality score Q is then defined as:

$$Q = -10 \log E \tag{3.11}$$

According to the characteristics of this definition, a high quality score Q indicates a small error probability, i.e. a read or variant call has been correctly determined. Conversely, low quality scores indicate a possibly erroneous read or variant call. Table 3.2 shows a range of sample Phred-scaled quality scores and their associated error probability. As can be seen, a Phred quality score ranging between 20 and 50 yields an accuracy between 99% and 99.999%.

Analysis

As we explore the feasibility of implementing posit arithmetic in the pair-HMM forward algorithm, it is of interest to compare the accuracy of the conversion of Phred-scaled scores to error probabilities for both the posit number format and for the IEEE 754 floating point type.

Q	**E**		**1 − E**
10	1/10	10%	90%
20	1/100	1%	99%
30	1/1000	0.1%	99.9%
40	1/10000	0.01%	99.99%
50	1/100000	0.001%	99.999%

Table 3.2: Example Phred-scaled quality scores and their associated error probability.

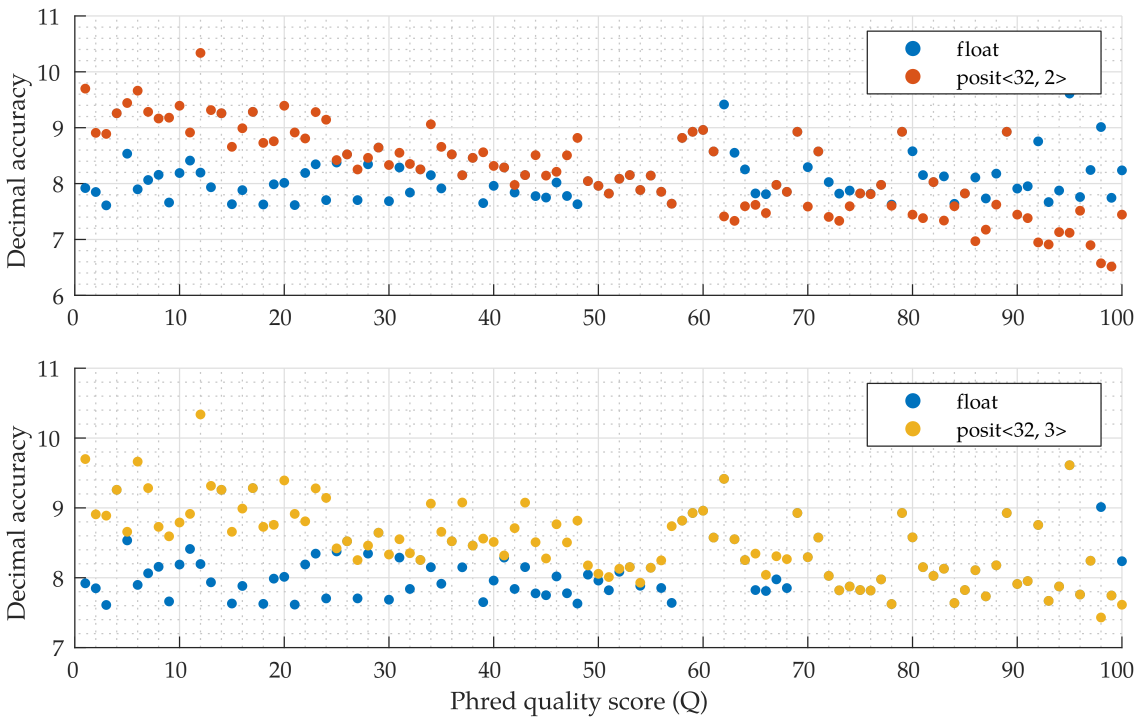


Figure 3.6: Decimal Accuracy for the conversion to error probabilities of Phred-scaled quality scores ranging from 1 to 100 for `float`, `posit<32,2>` and `posit<32,3>`.

Fig. 3.6 shows the decimal accuracy when converting a Phred quality score ranging from 1 to 100 to an error probability through the formula $P = 10^{-Q/10}$. The reference calculation performed in order to determine the decimal accuracy of the different number formats is performed in a 100-decimal accuracy number format using the Boost Multiprecision C++ library, providing a number type with a customizable number of decimal digits of precision at compile-time [35]. The decimal accuracy of these conversions are compared for the `float` format against the `posit<32,2>` and `posit<32,3>` posit formats respectively. As can be seen, the conversion of Phred scores to error probabilities using the `posit<32,2>` format results in approximately one more decimal of accuracy for scores ranging from 1 to 40, with decimal accuracy performing similar or worse in the range between 60 and 100. Since a typical score of a read ranges between 2 and 40 [36], these results prove that the posit format with 32 bits and 2 or 3 exponent bits could be an effective alternative to floats in order to represent error probabilities associated with the Phred quality score.

3.3.3 Pair-HMM Forward Algorithm

In order to explore the feasibility of applying posit arithmetic in the application of the pair-HMM forward algorithm (Section 3.1.1) and to benchmark its decimal accuracy by comparing to traditional IEEE 754 floating point calculations, a software implementation of the pair-HMM forward algorithm is developed that is able to perform all arithmetic operations using posit arithmetic using a well-established external library for posit number emulation [34]. A pseudo-code of this implementation is depicted in Appendix A. The source code is hosted open source [37]. The pair-HMM forward algorithm application is also benchmarked for the IEEE 754 32-bit `float` format in order to compare computation results for posits against regular floats.

Note that the test dataset for the analyses performed in the next sections are based on randomly generated emission and transmission probabilities (refer to Section 3.1). In order to obtain a fair benchmark between the `float`, `posit<32,2>` and `posit<32,3>` number formats, the probabilities for this test case are generated in such a way that the exact representation of each number is equal for all three number formats. The benchmark can be considered fair in the sense that different number representations should not have a head start during a decimal accuracy measurement for a specific application (in this case, the pair-HMM forward algorithm). Any head start could potentially be caused by the fact that one representation might be better at representing a specific random number compared to other representations. By ensuring the randomly generated number can be exactly represented in every candidate number format, accuracy loss caused by the representation of initially random numbers is omitted.

Initial Scaling Constant Dependency

As described in Section 3.1.1, a computational implementation of the pair-HMM forward algorithm often scales input values with a fixed initial constant in order to prevent un-

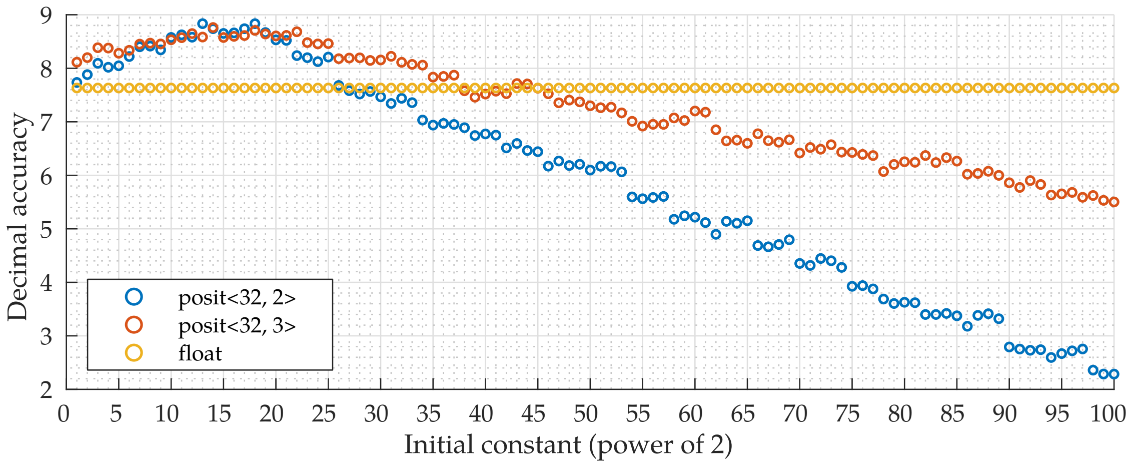
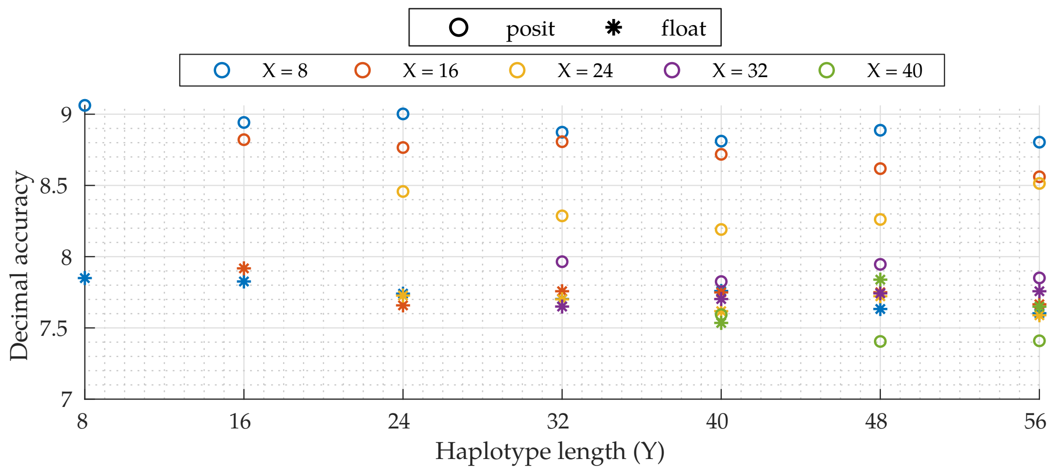


Figure 3.7: Comparison of decimal accuracy for the evaluation of the pair-HMM forward algorithm for different values of the initial scaling constant.

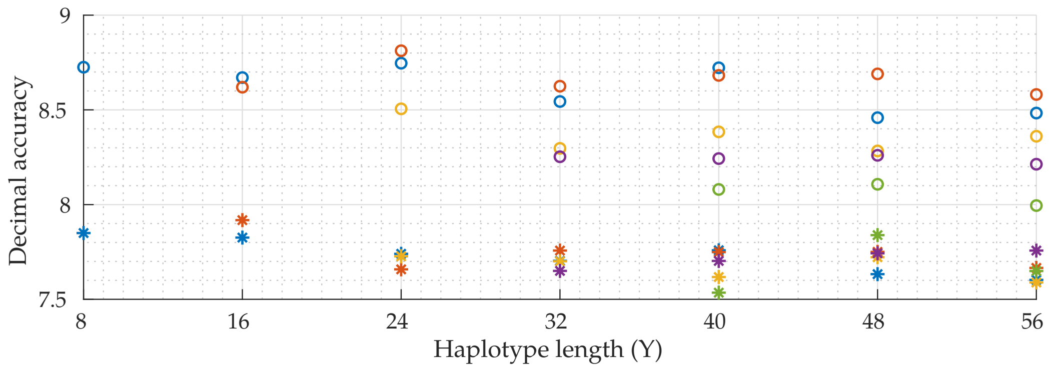
derflow in intermediate stages of computation. In order to determine the value of these initial constants, one has to take into account the number format that is used to represent intermediate numbers.

Fig. 3.7 shows the decimal accuracy when computing the likelihoods through the pair-HMM forward algorithm with the `float`, `posit<32,2>` and `posit<32,3>` number formats for a range of different initial constants that are used to initialize the computation matrices. This figure illustrates that the IEEE `float` format achieves a stable decimal accuracy of approximately 7.6 for this range of initial constants and for this particular test case. From the benchmark we can deduce that `posit<32,2>` achieves better decimal accuracy for all initial constants lower than approximately 2^{30} when comparing with `float`. Furthermore, the `posit<32,3>` type (where we increased the number of dedicated exponent bits by one) outperforms the `float` type in terms of accuracy for initial constant lower than 2^{40}. For both posit number configurations, the decimal accuracy is improved by 1.5 decimals in the most optimal case for an initial scaling constant of approximately 2^{15}.

The reason for the reduced decimal accuracy for higher initial constant values can be explained as follows. In the example above, the traditional `float` format is replaced by a 32-bit posit number type with, for example, 2 exponent bits. Assume an initial scaling constant set at 2^{100}. In order to convert an initial constant set at 2^{100} to a `posit<32,2>` environment, we calculate how many regime bits are required. Since $2^{100} = 16^{25} = \mathrm{useed}^{25}$ (for $\mathrm{useed} = 2^{2^{\mathrm{es}}} = 2^{2^2} = 16$), this value fits entirely in the regime field. In order to encode the value 25 in the regime, 27 regime bits are needed (refer to Section 2.1.3). Taking into account the remaining bits needed for the sign and exponent, this will leave us with only 2 bits left for the fraction. Naturally, this will result in poor decimal accuracy of any further calculations using this number.



(a) `posit<32,2>`



(b) `posit<32,3>`

Figure 3.8: Decimal accuracy of pair-HMM forward algorithm calculations for `posit<32,2>` and `posit<32,3>` compared to the decimal accuracy of calculations with the `float` type. X and Y denote the read and haplotype input sequence lengths respectively.

Overall Decimal Accuracy

Fig. 3.8 shows the decimal accuracy for the pair-HMM likelihood computation for different lengths of the read (X) and haplotype (Y) sequences. As can be seen, the decimal accuracy of the computation results is dependent on the input sequence lengths. This can be explained by the fact that longer input data sequences result in longer chains of calculations that depend on previous calculations. Therefore, the decimal accuracy generally decreases for longer calculations. Typical values for the length of read sequence lengths (X) average around 35 base pairs per read sequence [38]. Therefore, on the basis of the performed evaluation there is reason to believe that the posit number format could indeed be a worthy successor to the traditional IEEE 754 floating point format, costing the same number of bits, for evaluating the pair-HMM forward algorithm.

Chapter 4

Framework for Hardware Posit Arithmetic

The existing efforts to implement posit arithmetic hardware do not fulfill our requirements for applying posit arithmetic in hardware accelerator designs, as discussed in Section 2.3. For example, recall that existing units either (1) were not optimized for providing accurate answers by, for example, not applying any rounding scheme, or (2) were not designed for high-speed applications as the implementation consists of a single combinational path.

The aforementioned limitations in the state-of-the-art in hardware implementations for posit arithmetic lead to the proposal of a novel posit arithmetic framework that is designed to explore the potential use cases of applying posit arithmetic in hardware accelerators. This framework can be used to perform posit arithmetic in hardware without intermediate normalization of computation results, resulting in more accurate final answers. The posit adder, multiplier and accumulator units presented in this work therefore take the characteristic fields that represent a posit number as input instead of a serialized posit word. A posit normalization unit is able convert the unrounded regime, exponent and fraction fields to a traditional posit word whenever desired. Furthermore, a rounding scheme is applied in order to take into account bits that are truncated in the normalization process. In order to integrate posit arithmetic units into high-speed accelerators, the arithmetic units are pipelined such that the arithmetic units do not violate any timing constraints imposed by the interface between the host and accelerator.

This chapter is structured as follows. In Section 4.1 we will discuss the overall design of the arithmetic framework, including a detailed description of its individual components. The hardware implementation of the described framework in reconfigurable logic is discussed in Section 4.2, followed by an evaluation of the accuracy of the calculation results and its performance in Section 4.3. This chapter is concluded with a recap on the achieved results and relevant recommendations are discussed in Section 4.4.

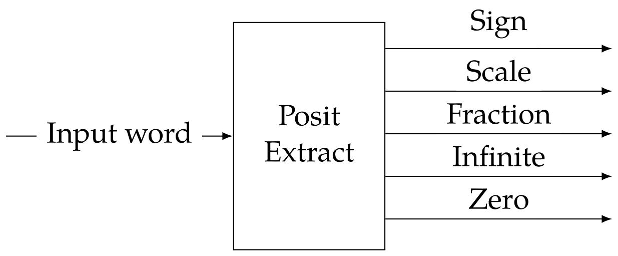


Figure 4.1: Schematic overview of the in- and output products for the proposed posit extraction unit.

4.1 Framework Design

The posit arithmetic framework presented in this work is built around the following three steps:

1. *Extraction*: extract the posit characteristics (sign, regime, exponent, fraction, infinite/zero) from an N-bit input word
2. *Operation*: perform an operation on the extracted posit fields
3. *Normalization*: normalize the output posit fields back into an N-bit posit word

The advantage of separating the extraction, operation and normalization steps becomes apparent when performing multiple arithmetic operations on a posit number, without having to normalize the result after every operation. Therefore, any loss of precision is averted. Based on this advantage, we propose a posit adder, accumulator and multiplier that take extracted posit fields as an input instead of an N-bit posit word that needs to be extracted (or unpacked) first. This allows us to feed in results from a previous addition or multiplication without having to normalize back and extract again the intermediate results.

In the next parts of this section, we will discuss the individual steps of the data flow described above, along with the behavior of the supported posit operations that are implemented.

4.1.1 Posit Extraction

The posit extraction unit converts an N-bit posit word to a data structure containing the characteristic fields of the posit operand. For both input operands the sign, *scale* and fraction bits are extracted. The scale serves as a scaling factor that is calculated based on the regime and the exponent of the input operands. Since the regime value accounts for a factor of $(2^{2^{\text{es}}})^{\text{regime}}$ (where *es* denotes the configurable size of the exponent field) and the exponent value adds a factor of 2^{exponent}, combining these factors results in the following scale factor for a `posit<nbits,es>` number [34]:

$$\text{scale} = (2^{2^{\text{es}}})^{\text{regime}} \times 2^{\text{exponent}} = 2^{2^{\text{es}} \times \text{regime} + \text{exponent}} \tag{4.1}$$

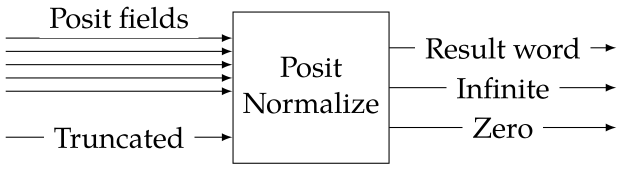


Figure 4.2: Schematic overview of the in- and output products for the proposed posit normalization unit.

For storing the separate fields of a posit, the base of 2 is made implicit. Therefore, the scale value represents the value $2^{\text{es}} \times$ regime + exponent. Since the maximum value represented by the regime field is equal to $\text{nbits} - 2$ and $2^{\text{es}} - 1$ for the exponent, the total number of bits required to represent any scale is equal to

$$\begin{aligned} \text{scale width} &= \lceil \log_2 (2^{\text{es}} \times (\text{nbits} - 2) + 2^{\text{es}} - 1) \rceil && (4.2) \\ &= \lceil \log_2 (2^{\text{es}} \times (\text{nbits} - 1) - 1) \rceil && (4.3) \end{aligned}$$

Based on this, in order to represent the full characteristics of a posit value, the *sign*, *scale*, *fraction* and *infinite/zero* status flags are extracted. The overall schematic overview of the posit extraction module is depicted in Fig. 4.1. The bit field widths of each field for the different values are derived in Appendix B.

4.1.2 Posit Normalization

In order to convert the internal structure of posit fields back into a regular posit word (including the unrounded fraction), normalization needs to be performed.

Normalization of an addition, accumulation or multiplication result induces a loss in decimal accuracy as a part of the fraction field is truncated. Since the fraction of an N-bit posit number can be up to $(\text{N} - \text{es} - 2)$ bits wide (including hidden bit), integer addition or multiplication of the two operand fractions results in a larger fraction. Therefore, this number will be truncated when included in the final N-bit product posit. The number of bits that are truncated depends on the number of bits discarded in order to represent the regime and the exponent in the final posit number.

In order to preserve as much decimal accuracy as possible, a rounding scheme is implemented. The rounding scheme implemented for the proposed arithmetic units is *round to nearest, tie to even*. This rounding scheme is chosen as being the only rounding mode available for the posit scheme [7], and is chosen as default rounding scheme for the IEEE float format. Rounding is thus performed to the nearest value. Consequently, all fraction bits that are discarded (in order to fit the final result in an N-bit value) are used to determine whether a value has be rounded. For a tie (midway value), the value is rounded to an even number, i.e. the Least Significant Bit (LSB) is zero. The posit adder and accumulator units, described later in this section, are able to assert a *truncated* flag whenever bits need to be truncated in order to match the scales of both input operands. This flag is also taken into account by the rounding scheme implemented in the posit normalization unit.

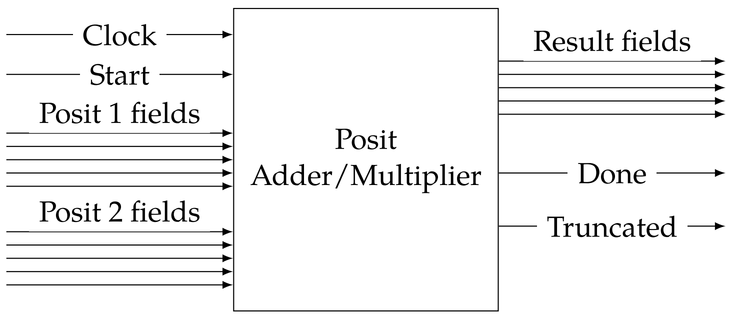


Figure 4.3: Schematic overview of the in- and output products for the proposed posit adder and multiplier.

The scale field of the input posit structure is used to determine the regime and exponent for the resulting N-bit posit number. The regime is calculated as scale/2^{es}, while the exponent is determined by the remainder (scale $\bmod 2^{\text{es}}$). The combination of exponent and fraction is shifted right by the amount of bits needed to represent the final regime. After the packed regime, exponent and fraction have been obtained, rounding is performed when needed by adding 1 LSB to this result. Finally, in case the sign of the final posit number is negative, the 2's complement of the regime, exponent and fraction are determined.

4.1.3 Posit Adder

After normalization of any N-bit input operands, the extracted posit fields can be passed to a posit adder, along with a start signal that validates the output results as they are propagated through the adder. A summary of the step-by-step operations in the posit adder design is as follows:

1. As a preparation before performing the actual posit addition, the hidden bit is prepended to the input fraction fields.
2. In order to match the scale (as defined in Eq. (4.1)) of both operands, the smallest operand needs to be shifted right in order to match the scale of the largest operand. Therefore, the largest and smallest operands are determined. The smallest operand fraction is shifted right by the difference between both operand scales.
3. In the aforementioned process, the fraction field of the smallest operand might lose bits due to the shifting performed in order to match both operand scales. A *truncated* flag is asserted by the adder that can be used by the normalization unit when performing the rounding of the truncated sum fraction (described in Section 4.1.2) whenever a result needs to be normalized.
4. Both operand fractions are added (or subtracted for unequal operand signs) using an unsigned integer adder.

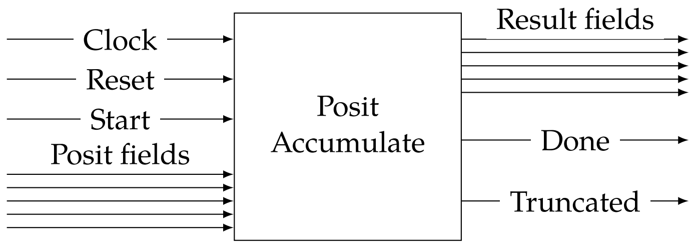


Figure 4.4: Schematic overview of the in- and output products for the proposed posit accumulator.

5. After detecting the location of the hidden bit in the sum, the fraction sum is normalized by shifting left until the normalized form `1.xxx` is reached. The sum scale, which is set at the scale of the largest input operand, is updated accordingly.

The resulting posit sum, consisting of a structure of the unrounded scale and fraction fields (among others), can then be used as an input operand for next operation(s) or can be normalized back into a regular N-bit posit word through the posit normalization unit described in Section 4.1.2.

4.1.4 Posit Multiplier

Similar to the posit adder, the posit multiplier unit performs a multiplication operation on two posit operands, producing a set of fields which represents the unrounded posit product. A summary of the operations for the posit multiplier is as follows:

1. The fraction field of the input posit operands are multiplied using an unsigned integer multiplier.
2. The scale of the output product, as defined in Eq. (4.1), is determined by adding the scales of both input operands. This scale is increased by 1 in case of an overflow in the fraction multiplication.
3. The resulting product fraction is shifted in order to obtain the normalized form `1.xxxx`.

The posit multiplier returns a structure of the unrounded scale and fraction fields, among others. This structure can be used as an input operand for next operation(s) or can be normalized (or packed) into a regular N-bit posit number using the proposed posit normalization unit.

4.1.5 Posit Accumulator

The posit adder described in Section 4.1.3 is designed to calculate the sum of two N-bit input posit words. Recall that the fraction of the smaller input operand is shifted in order

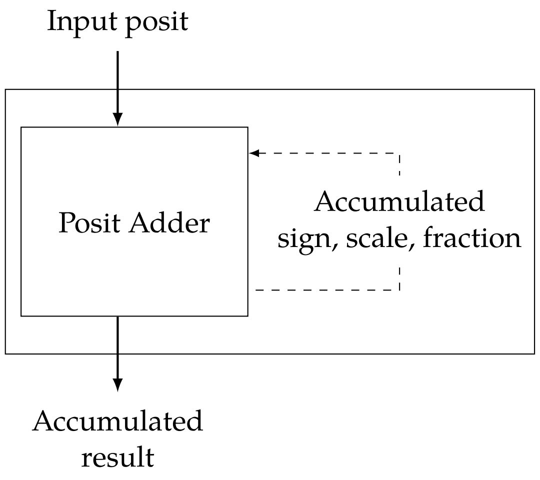


Figure 4.5: Schematic overview of the posit wide accumulator, consisting of a posit adder module with the accumulated looped back as an input operand.

to match the scale of both input operands. It is therefore possible that one or multiple fraction bits of the smaller operand are discarded before the fraction addition step is being performed. This is undesirable when designing an implementation that is optimized in terms of achievable decimal accuracy.

In order to avoid any input information to be discarded, the posit wide accumulator as depicted in Fig. 4.4 is proposed. The wide accumulator consists of a posit adder that is similar to the design proposed in Section 4.1.3, but differs in the fact that the adder only takes one posit input. The second input consists of the sign, scale and fraction of the accumulated number so far, which is the result of the accumulator looped back to the input port. This is illustrated in Fig. 4.5. Note that the looped back accumulated fraction is not normalized and consequently contains all bits resulting from the sum with the input posit operand, preserving the input information. The output of the wide accumulator consists of the posit fields of the accumulated number. This output can, for example, be directed to a posit normalization unit (refer to Section 4.1.2) to obtain a regular N-bit posit number.

4.2 Hardware Implementation

The design of the novel posit adder, accumulator and multiplier as described in Section 4.1 are implemented in reconfigurable hardware for the `posit<32,2>` and `posit<32,3>` configurations. The target FPGA for this implementation is the Xilinx Kintex® UltraScale™ XCKU060. Area utilization for the different implementations can be found in Table 4.1. For the accumulator, two separate designs are implemented in order to accumulate either regular values, or values coming from a multiplier (which have twice as many fraction

Implementation	`posit<32,2>`		`posit<32,3>`	
	LUTs	**Registers**	**LUTs**	**Registers**
Extractor	236	0	236	0
Adder (4-stage)	331	241	325	245
Adder (8-stage)	385	287	373	374
Accumulator (Regular)	2311	2437	4014	4029
Accumulator (Product)	2409	2479	4252	4070
Multiplier (4-stage)	80	254	78	252
Normalizer (Regular)	246	0	218	0
Normalizer (Accumulated Value)	290	0	905	0
Normalizer (Accumulated Product)	306	0	1090	0
Standalone Adder (4-stage)	1151	230	1141	262
Standalone Adder (8-stage)	1095	382	1095	380
Standalone Multiplier (4-stage)	963	225	864	226

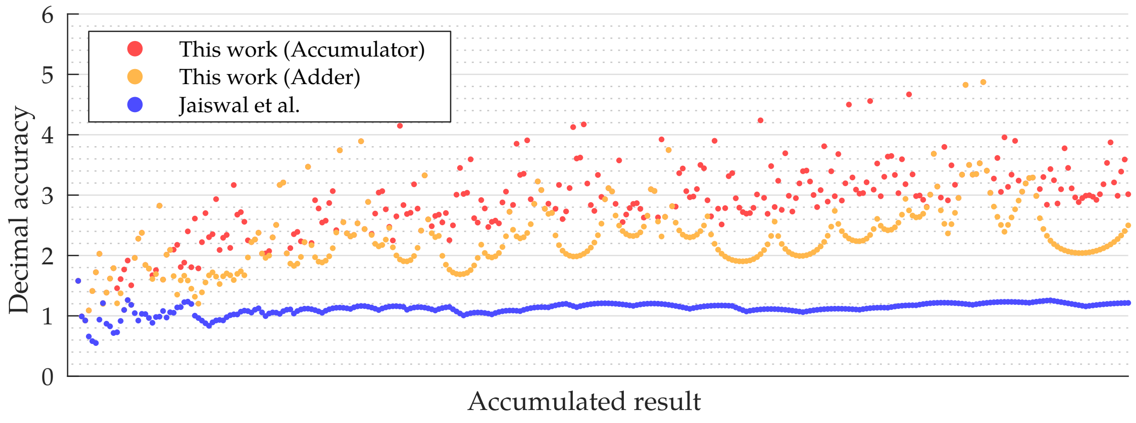
Table 4.1: Area utilization for the implementation of the posit adder, accumulator and multiplier designs discussed in Section 4.1.

bits compared to a regular posit number). Multiple implementations are also considered for the posit normalizer unit as different inputs can be accepted, such as a regular posit value, an accumulated value or a product. As there are use cases where a single operation is performed on arithmetic numbers, standalone posit arithmetic units are also considered. These standalone units consist of the extraction, operation and normalization steps combined in one unit.

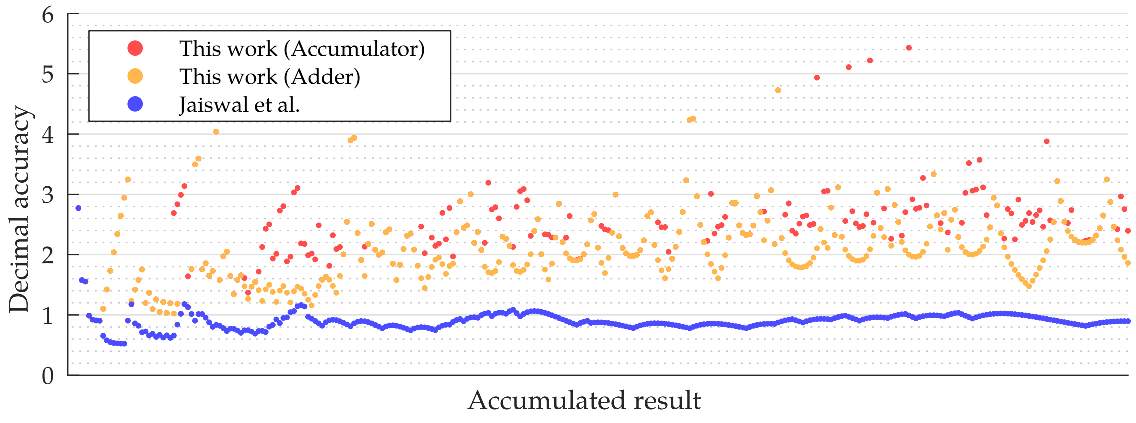
One of the conclusions that can be made from the utilizations reported in Table 4.1 is the fact that the accumulator implementation LUT usage is approximately $6.5\times$ and $12.5\times$ higher compared to the regular adder design for 2 and 3 exponent bits respectively. This can be explained by the fact that in this design, wider fraction fields are maintained while accumulating (refer to Appendix B), without truncating any intermediate result. Therefore, a tradeoff exists between the amount of area used and the degree of precision that is required in a specific application. One could opt for a simple accumulator built using the regular posit adder design with the output looped back to one of the inputs instead. This would save on resources but gives in on achievable decimal accuracy, as will be shown in the next section.

4.3 Evaluation of Framework Accuracy & Performance

As the discussed posit arithmetic framework is designed with decimal accuracy in mind, a benchmark is performed in order to measure the amount of decimal accuracy achieved compared to related implementations. Furthermore, the performance achieved by the individual components presented in this framework is quantified.



(a) `posit<32,2>`



(b) `posit<32,3>`

Figure 4.6: Decimal accuracy of an accumulation of 300 incrementing fractional numbers for a 32-bit posit number format with 2 and 3 exponent bits, respectively.

4.3.1 Decimal Accuracy

We compare the results of the novel posit arithmetic unit designs with a reference calculation performed by software using the Boost multiprecision library [35], capable of performing calculations with at least 100 decimals of accuracy. In order to quantify the accuracy of the proposed arithmetic units, the definition of decimal accuracy as discussed in Section 2.6 is used. This calculation is also performed based on a 100-decimal precision reference calculation. Furthermore, the decimal accuracy results are evaluated for the accumulator described in this work (Section 4.1.5), as well as for an accumulator that is implemented using the posit adder design as presented by Jaiswal et al. [10] (discussed in Section 2.3). In order to perform a more ”fair” comparison, results are also shown for the accumulator built with a single posit adder unit from this work where extraction, addition and normalization are performed in each cycle. Hence, the wide unrounded fraction fields are omitted in this design.

Fig. 4.6a shows the decimal accuracy of an accumulation of the first 300 smallest representable fractions for a `posit<32,2>` configuration. As can be seen, the decimal accuracy of the accumulator built with the traditional adder presented in this work is improved by approximately 1 decimal of accuracy for the final (rightmost) result, when comparing to the accumulator built with the adder presented by [10]. Using the wide accumulator presented in this work results in an improvement of more than 2 decimals of accuracy compared to the results of the alternative implementation by [10], and a tight 1 decimal of accuracy improvement toward the adder presented in this work.

Similar to the results for a `posit<32,2>` configuration, the decimal accuracy of an accumulation of `posit<32,3>` fractions is depicted in Fig. 4.6b, showing similar improvement in decimal accuracy for the accumulator design presented in this work. It can be noted that the decimal accuracy results of the wide accumulator and the accumulator built with a regular posit adder lie closer towards each other for the `posit<32,3>` configuration compared to the previous results for `posit<32,2>`. This can be explained by the following observation. Increasing the number of exponent bits by one unit yields a quadratic increase of *useed*, i.e. changing the number of exponent bits from 2 to 3 changes *useed* from 16 to 256. Recall that the smallest representable number for a specific posit configuration is calculated as follows [2]:

$$\text{minpos} = \text{useed}^{2-n} = 2^{2^{\text{es}}(2-n)} \tag{4.4}$$

Based on this equation, switching from 2 to 3 exponent bits results in a change in the smallest representable number from approximately 10^{-37} to 10^{-73} for 32-bit posit numbers. At the same time, as bits are occupied in order to represent the exponent, less fraction bits are available. This reduces the achievable dynamic range of a specific posit configuration. It is trivial to see that these two implications are the explanation for the fact that less decimal accuracy is lost due to rounding errors (which could modify a calculation result by at most one bit), since one LSB will represent a considerably smaller number as the number of exponent bits in the posit configuration increases.

4.3.2 Performance

For each unit presented in this work, the maximum achievable performance in Posit Operations Per Second (POPS) is depicted in Table 4.2. The target FPGA used for these measurements is the Xilinx Kintex® UltraScale™ XCKU060 FPGA. The proposed implementations achieve a performance of approximately 250 MPOPS for addition, 160 MPOPS for multiplication and 180 MPOPS for accumulation operations. Although measures of performance are highly dependent on the target platform, the implementation in this work shows to be more suitable in high-speed applications compared to related work. For example, the posit arithmetic units proposed by Jaiswal et al. [10] have a latency of approximately $12.5\,\text{ns}$ (80 MPOPS) and $12.9\,\text{ns}$ (77.5 MPOPS) for addition and multiplication operations respectively for the same FPGA that is used for this implementation. From this we can deduce that the implementation presented in this work achieves a speedup of over $2\times$ compared to the latest related work in posit arithmetic hardware development.

Hardware Implementation	`posit<32,2>`	`posit<32,3>`
Adder (4-stage)	229.62 MPOPS	283.53 MPOPS
Adder (8-stage)	223.66 MPOPS	261.51 MPOPS
Accumulator (Regular)	186.36 MPOPS	171.44 MPOPS
Accumulator (Product)	186.36 MPOPS	175.04 MPOPS
Multiplier (4-stage)	158.05 MPOPS	166.20 MPOPS
Standalone Adder (4-stage)	150.46 MPOPS	151.29 MPOPS
Standalone Adder (8-stage)	170.42 MPOPS	162.28 MPOPS
Standalone Multiplier (4-stage)	158.83 MPOPS	152.51 MPOPS

Software Emulation	`posit<32,2>`	`posit<32,3>`
Addition	0.1006 MPOPS	0.1004 MPOPS
Multiplication	0.0177 MPOPS	0.0173 MPOPS

Table 4.2: Computation performance in MPOPS of the units presented in the posit arithmetic framework, as well as for posit operations computed by means of posit emulation through an external library for the platform described in Section 4.3.2.

Table 4.2 also shows performance measurements in software for performing posit arithmetic operations using an external posit emulation library [34]. These measurements have been performed on an Intel® Xeon® E5-2680 v4 CPU running at $2.4\,\mathrm{GHz}$ with $192\,\mathrm{GB}$ RAM and is averaged over 100.000 unique calculations. Based on these numbers, an average speedup of $2500\times$ is achieved for calculation in hardware versus an emulation through software. It is imperative to acknowledge the fact that this speedup calculation is only a rough estimate: a performance comparison between software and hardware is highly dependent on the target platform. However, these results do indicate the advantage of performing posit arithmetic in hardware opposed to performing only an emulation of the posit number representation system in software.

4.4 Summary

In this chapter we proposed a novel framework for performing posit arithmetic optimized for decimal accuracy. In order to perform high-precision posit operations, a posit extract module extracts, or unpacks, the properties of an input posit number. Consequently, addition and multiplication operations can be performed without rounding off intermediate results. Instead, a final posit normalization unit can be used in order to convert the internal memory layout of the unrounded result back to a regular posit number. This way, we minimize loss of decimal accuracy that might occur through intermediate calculations. Together with the aforementioned advantages, implementing the *round to nearest, tie to even* rounding scheme in the posit normalization unit of this work results in more accurate computation results as fraction bits that are about to be discarded in the final result are taken into account when determining the final posit word.

Based on the discussed results regarding accuracy and performance in Section 4.3.2, several conclusions can be made. Regarding decimal accuracy of calculation outcomes, a major improvement in decimal accuracy is observable when comparing with related work such as [10]. For example, it is shown that the implementation of the posit accumulator unit yields an improvement of approximately one decimal of accuracy compared to accumulation using the regular posit adder presented in this work. Furthermore, approximately two more decimals of accuracy are achieved when comparing our results to the alternative implementation presented by [10].

The implementation of the designs presented in this work achieve approximately 250 MPOPS for addition, 160 MPOPS for multiplication and 180 MPOPS for accumulation operations. Although measures of performance are highly dependent on the target platform, the implementation in this work shows to be more suitable in high-speed applications compared to related work where, for example, single-cycle implementations have been developed.

Although increased decimal accuracy is achieved, there are also limitations with respect to area usage. As discussed in Section 4.2, implementing a wide accumulator that is designed to accumulate posit numbers without intermediate fraction rounding results in a 6.5 to $12.5\times$ increase in area usage for 2 and 3 exponent bits respectively when comparing with the regular adder design presented in this work.

The added value of the proposed arithmetic framework becomes apparent in the next chapters where practical examples of the application of the presented posit arithmetic building blocks will be shown.

Chapter 5

Posit Vector Arithmetic Accelerator

In this chapter, we discuss the implementation of an accelerator for performing Level 1 BLAS operations on posit column vectors, connected through the Coherent Accelerator Processor Interface (CAPI) SNAP framework and able to read input column vector data using the Apache Arrow in-memory data platform. The Apache Arrow platform, format and its advantages and/or disadvantages are discussed in Section 2.7.

The main goal of the proposed posit arithmetic accelerator design is to accelerate common operations on column vectors, also known as Level 1 Basic Linear Algebra Subprograms (BLAS), for vectors consisting of posit numbers. Level 1 BLAS vector operations consist, among others, of the element-wise vector addition or subtraction and multiplication operations, including element-wise operations with a scalar operand, returning a result vector. Furthermore, common linear algebra operations are the vector dot product and vector sum aggregation operations that transform one or multiple (variable length) vectors into a single, scalar posit value.

First, we discuss the design of the posit arithmetic accelerator in Section 5.1. The synthesized implementation of this design, as discussed in Section 5.2, is then evaluated based on performance and accuracy in Section 5.3. This chapter is then summarized and future work regarding the implementation presented in this chapter is discussed in Section 5.4.

5.1 Accelerator Design

The proposed posit arithmetic accelerator is designed with modularity and usability in mind. As different vector operations are supported, the number of input vectors is variable, as well as the type of output, being either a scalar value or a vector. The different vector operations can be combined in order to implement a specific algorithm. An example algorithm (the Gram-Schmidt process) is discussed in Section 5.3.2.

The input posit vectors for this accelerator design are represented in the Apache Arrow in-memory format. Therefore, the accelerator should be able to retrieve the input

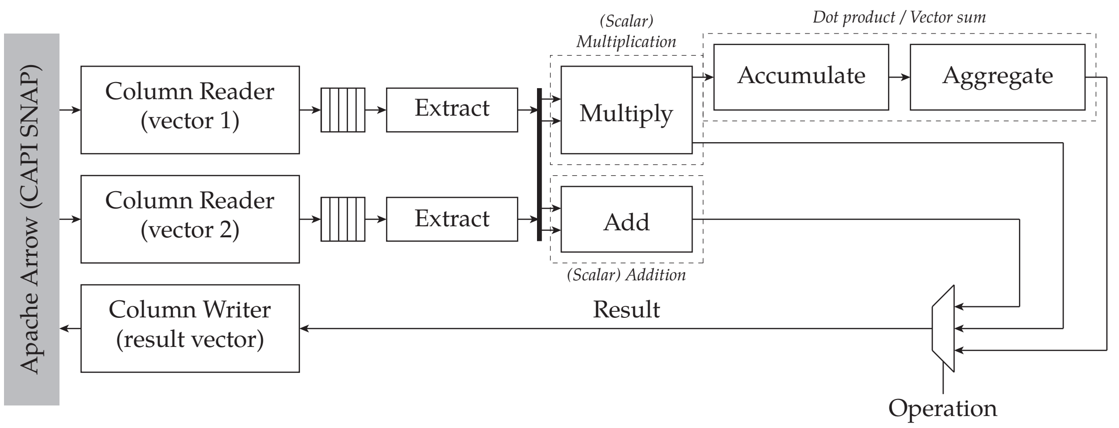


Figure 5.1: Schematic overview of the data flow for the posit vector arithmetic accelerator, starting from the input vector representations in Apache Arrow. Each component is annotated with the operations this component is used for.

vectors from this format. The CAPI SNAP platform is used in order to interface between host memory and the accelerator. In order to retrieve data stored in the Apache Arrow format, the Fletcher framework is used [39]. This framework is designed for integrating FPGA accelerators with Apache Arrow. For the posit arithmetic operations performed in this accelerator, the arithmetic framework as described in Section 4.1 is implemented.

A schematic overview of the accelerator structure is depicted in Fig. 5.1, along with annotations of which components are used for each supported vector operation. The general behavior of the accelerator during the process of performing a vector operation can be described as follows. For each input vector, one column reader is instantiated in order to read the input vector elements. These column readers are provided by the Fletcher framework [39]. A column reader enables reading from an Apache Arrow column buffer. As the accelerator is designed to accept two input vectors, one column reader is instantiated per posit input vector. In this design, the vector elements are stored into one FIFO component for each vector, which has two advantages. First, we are able to keep receiving vector elements while performing the desired vector arithmetic operation. Secondly, the posit arithmetic units are fed with one unique input element every cycle. Hence, the FIFOs act as a buffer to compensate for the potential irregular output of the column readers.

As part of the initialization of the accelerator, the Arrow buffers containing the input element vectors are initialized with the input data. Consequently, the addresses of these buffers are communicated to the accelerator through Memory Mapped Input/Output (MMIO) registers. The column readers are then able to access the element vectors located inside these buffers. After instantiation of the Arrow buffers containing the input vectors for the accelerator, the host is able to start the accelerator. Subsequently, the accelerator FIFOs are filled with vector elements produced by the column reader. The FIFO outputs are connected to posit extraction units, extracting the fields of the input posit elements (refer to Section 4.1).

Depending on the desired vector arithmetic operation, the extracted posit fields of both vector elements serve as operands to a posit multiplier or adder. For a vector addition/subtraction or multiplication, the resulting sum or product directly serves as one of the elements in the final result vector (see Fig. 5.1). In case of a posit dot product calculation, the unrounded multiplication result is passed to an accumulator unit that is able to accumulate unrounded posit products (Section 4.1.5). This process is repeated until all elements of both vectors have been processed. If this is the case, the final aggregation of accumulated posit products will start. This step is necessary, as the posit accumulator used in this design is pipelined with 16 stages. Thus, 16 individual accumulations are maintained, accepting new input posits every cycle. These 16 accumulations are aggregated by another posit accumulator serving as aggregation unit. This process of calculating the dot product of two input vectors can also be used to calculate the sum of a single vector. In this case, the second input vector is set to a vector of ones.

Note that the resulting values coming from the adder, multiplier or accumulator consist of an unrounded set of posit fields (as described in Section 4.1.1). These values are normalized to a regular N-bit posit in the final stage of the accelerator. The elements of the resulting posit vector (or single value for a dot product or vector sum calculation) are written back to host memory, represented by an Apache Arrow buffer using a column writer.

As discussed in the beginning of this section, the accelerator has been designed with modularity taken into account. Therefore, a software library (C++) is developed in order to easily interface with the proposed accelerator. This library provides the following functions to the programmer in order to perform a specific vector operation.

- `vector_add`: Element-wise vector addition
- `vector_sub`: Element-wise vector subtraction
- `vector_mult`: Element-wise vector multiplication
- `vector_dot`: Vector dot product
- `vector_sum`: Vector sum

The provided library serves as a drop-in replacement for existing software routines for performing (posit) vector arithmetic. The described functions prepare any input data that is not yet represented in the Apache Arrow columnar memory format (Section 2.7) by transforming it into this format. Furthermore, the desired vector operation to be executed will be communicated to the accelerator, after which the addresses to the Arrow buffers containing the input vectors is transmitted. The library will handle the buffering of the final result vector (or scalar) and provides it to the user for any further processing.

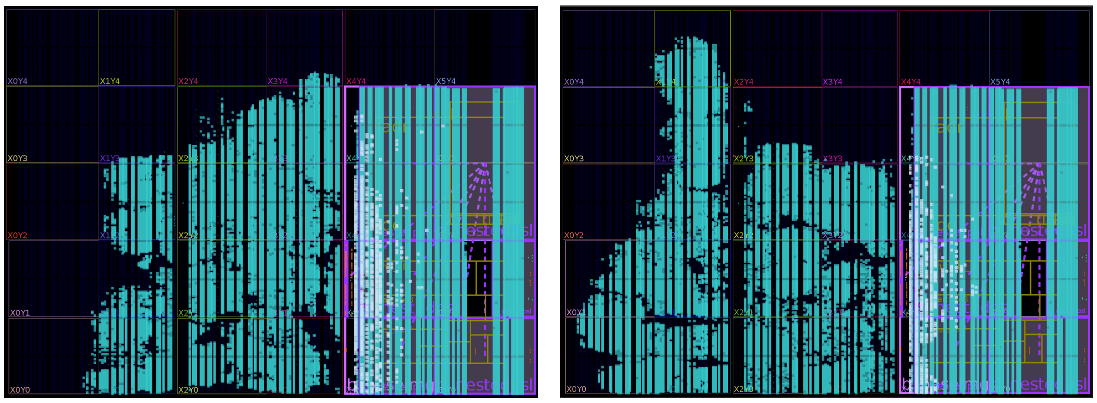

(a) es = 2 (b) es = 3

Figure 5.2: Post-route layouts of the posit vector arithmetic accelerator interfacing with Apache Arrow in-memory data using the CAPI SNAP platform. FPGA device: Xilinx UltraScale XCKU060.

5.2 Hardware Implementation

In this section we discuss the implementation of the accelerator design as discussed in Section 5.1. An implementation has been generated and tested for a `posit<32,2>` and `posit<32,3>` configuration. The target FPGA for this implementation is the Xilinx Kintex® UltraScale^{TM} XCKU060 FPGA. The working frequency of this implementation is 125 MHz. The post-route layouts of these implementations are depicted in Fig. 5.2.

Table 5.1 shows the area utilization statistics for the posit dot product accelerator implementation as well as the estimated power consumption. The area usage of both the accelerator core only as well as for the total design is displayed. The overall design includes the implementation of the Power Service Layer (PSL), required for interfacing with the host using CAPI. Overall, approximately 40% of the available FPGA resources are used for this design.

5.3 Evaluation of Performance & Accuracy

For the implementation as described in Section 5.2, we evaluate the performance of the implementation in terms of performance and decimal accuracy of the accelerator calculation results. In order to quantify the performance of the proposed accelerator, we compare the execution time of the hardware accelerator versus the same calculation in software, using the most popular library used for emulating the posit number format [34]. Note that the execution time of the benchmark calculations performed in software are highly dependent on the specifications of the machine used. The machine used in this experiment is the IBM® Power Systems^{TM} S822LC featuring two 10-core POWER8 CPUs running at 2.92 GHz.

Configuration		**Available**	**Used (core)**		**Used (total)**	
`posit<32,2>`	**LUT**	331680	52265	(15.76%)	131189	(39.55%)
	Register	663360	64969	(8.64%)	156747	(23.63%)
	BRAM	1080	91	(9.79%)	417	(37.18%)
	DSP	2760	4	(0.14%)	23	(0.83%)
	Power		2.495 W		9.543 W	
`posit<32,3>`	**LUT**	331680	52262	(15.76%)	131179	(39.55%)
	Register	663360	65033	(8.29%)	156827	(23.64%)
	BRAM	1080	91	(8.43%)	417	(38.61%)
	DSP	2760	4	(0.14%)	23	(0.83%)
	Power		2.294 W		9.358 W	

Table 5.1: FPGA resource utilization and power consumption estimation of the posit dot product accelerator implementation, both for the accelerator core only and for the total implementation including the Power Service Layer.

5.3.1 Vector Operations

As multiple vector operations are supported by the proposed accelerator, each operation is benchmarked based on its performance and speedup compared to software calculation. Furthermore, in order to illustrate the average accuracy achieved by the discussed accelerator, will evaluate the decimal accuracy of the posit dot product operation performed by the accelerator.

Performance

Fig. 5.3 shows the execution times for both hardware and (single thread) software implementations for the calculation of the dot product of two input posit vectors, for an increasing number of input vector lengths. The measured hardware execution time includes the overhead caused by the initialization of the hardware device and the reading and writing of the in- and output data respectively. The right axis shows the speedup of the hardware implementation compared to software. As can be seen, the speedup of the hardware implementation compared to the software implementation is dependent on the input vector lengths. For an input vector length of 10^6 posit elements, a speedup of approximately $15000\times$ is achieved. The speedup for smaller input vector sizes is absent or relatively low for small input vector sizes. This can be explained by the accelerator overhead. The accelerator overhead can be seen at the execution time for an input vector length of 1 element, which effectively reduces the dot product calculation to a single multiplication operation. For larger input vector sizes, starting around 10^5, the slope of the speedup curve is reduced. This can be explained by hardware saturation: the bandwidth of the accelerator becomes limited by the input buffer FIFOs of the accelerator design.

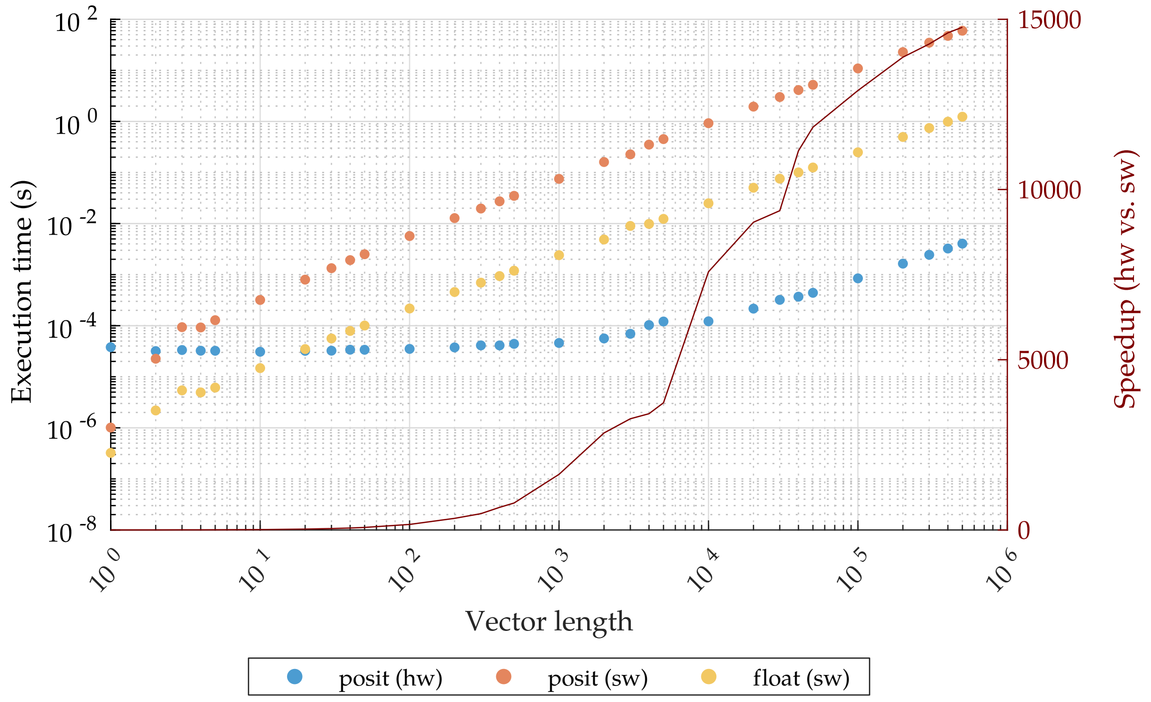


Figure 5.3: Execution time for a posit dot product calculation using the proposed hardware accelerator compared to dot product calculation in software (single-thread) for different input vector lengths. The curve illustrates the speedup of the hardware posit implementation compared to software posit emulation.

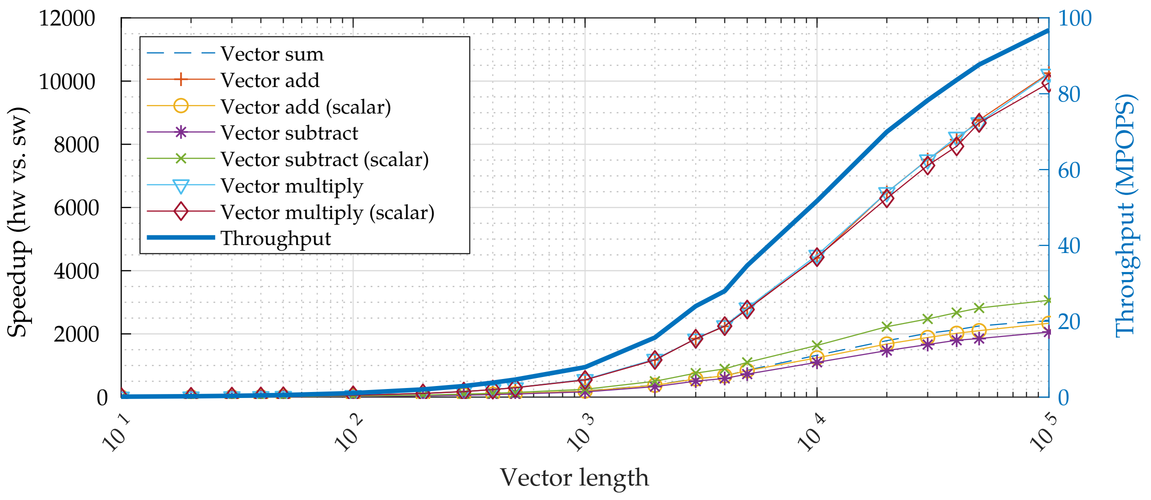
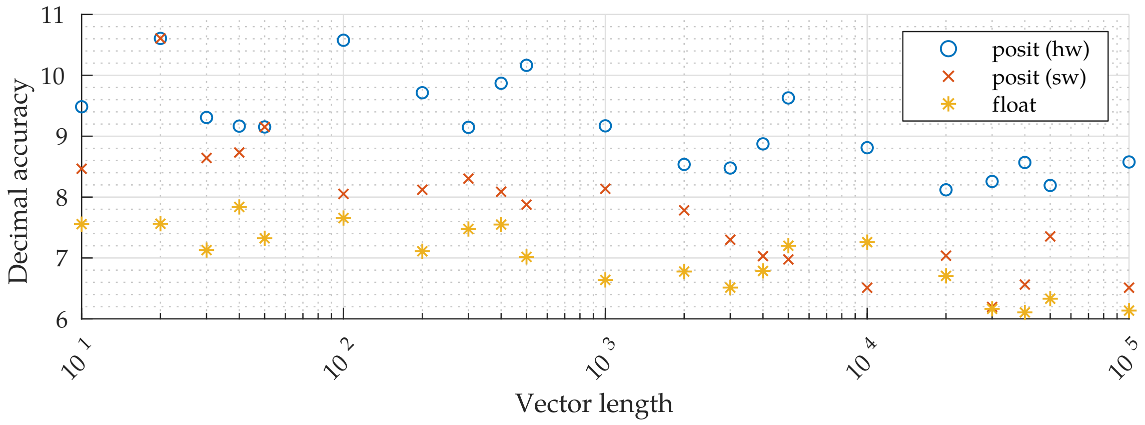
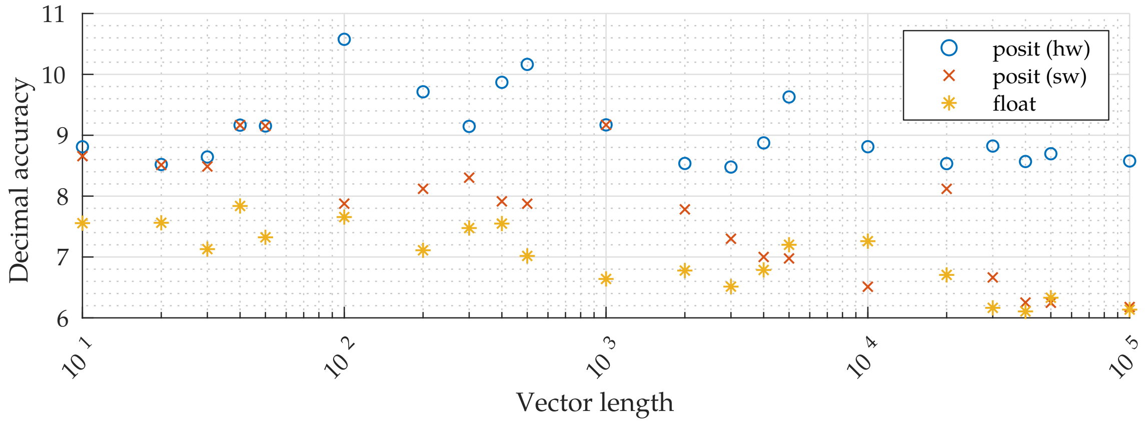


Figure 5.4: Speedup of the posit vector arithmetic accelerator for different vector operations and vector input lengths, compared to the execution time for posit emulation in software. The hardware throughput (in MPOPS) is depicted on the right axis.



(a) `posit<32,2>`



(b) `posit<32,3>`

Figure 5.5: Decimal accuracy of calculation results of the posit dot product operation performed by the proposed posit vector arithmetic accelerator and compared to software calculation results through software emulation, for different input vector lengths.

Fig. 5.4 shows the speedup compared to software calculation of the other vector operations supported by the proposed accelerator. The accelerator throughput in Posit Operations Per Second (POPS) is depicted on the right axis (bold line). As can be seen, especially the element-wise vector multiplication operations benefit from hardware acceleration with around $800\times$ speedup for a vector length of 10^3 elements. For this input vector length, other operations benefit from acceleration as well by achieving a speedup of over $100\times$.

Decimal Accuracy

As described in Section 5.2, implementations of the posit vector arithmetic accelerator have been generated for the `posit<32,2>` and `posit<32,3>` configurations. Fig. 5.5 shows the decimal accuracy of the posit dot product calculation results produced by the accelerator for both posit configurations. The decimal accuracy is compared to software calculation in the posit and IEEE 754 float format for a range of input vector lengths. The input vectors for this test case consist of randomly generated values. Similar to the analysis performed for the pair-HMM forward algorithm in Section 3.3.3, the generated values for this test case are created such that the exact representation of each number is equal for all three number formats. Hence, an improved decimal accuracy of one number format compared to another is not possible at the initial stage of computation.

As can be seen, for both posit configurations, the decimal accuracy of the hardware results are improved by approximately one decimal of accuracy compared to calculation results by software (through emulation of the posit number format). Furthermore, an increase of approximately two decimals of accuracy is achieved compared to calculation using the traditional IEEE 754 floating point format.

5.3.2 Gram-Schmidt Process

Among the potential applications of the proposed posit vector arithmetic accelerator is the Gram-Schmidt process. The Gram-Schmidt process is a common method for orthogonalization of a set of vectors. Using the capabilities of the posit arithmetic accelerator discussed in this chapter, it is possible to implement a hardware accelerated version of the Gram-Schmidt process using posit arithmetic.

Given a set $S = \{\sigma_1, \dots, \sigma_k\}$ consisting of k vectors from the vector space $\mathbb{R}^n$, the Gram-Schmidt process generates a set of vectors $\{u_1, \dots, u_e\}$ $(e \leq k)$ which are pairwise orthogonal, i.e.

$$\langle \sigma_i, \sigma_j \rangle = \delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

and form the basis of a subspace S' of $\mathbb{R}^n$ [40].

Define the projection operator $\mathrm{proj}_{\mathbf{u}}(\mathbf{v})$ that performs the projection of a given vector $\mathbf{v}$ onto $\mathbf{u}$:

$$\mathrm{proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\langle \mathbf{u}, \mathbf{u} \rangle}\mathbf{u} \tag{5.2}$$

where $\langle \mathbf{u}, \mathbf{v} \rangle$ denotes the inner product between $\mathbf{u}$ and $\mathbf{v}$.

Then, the Gram-Schmidt sequence $u_1, \dots, u_k$ is calculated as

$$\begin{aligned} \mathbf{u}_1 &= \mathbf{v}_1 \\ \mathbf{u}_e &= \mathbf{v}_e - \sum_{j=1}^{e-1} \mathrm{proj}_{\mathbf{u}_j}(\mathbf{v}_e) \qquad \text{for } e = 2, \dots, k \end{aligned} \tag{5.3}$$

The obtained Gram-Schmidt sequence forms a set of vectors orthogonal to each other. By normalizing the obtained set of vectors $\mathbf{u}$, an orthonormal set can be obtained. The Gram-Schmidt process has an application in the QR-decomposition of matrices, which is the process of decomposing a given matrix A into

$$A = QR \tag{5.4}$$

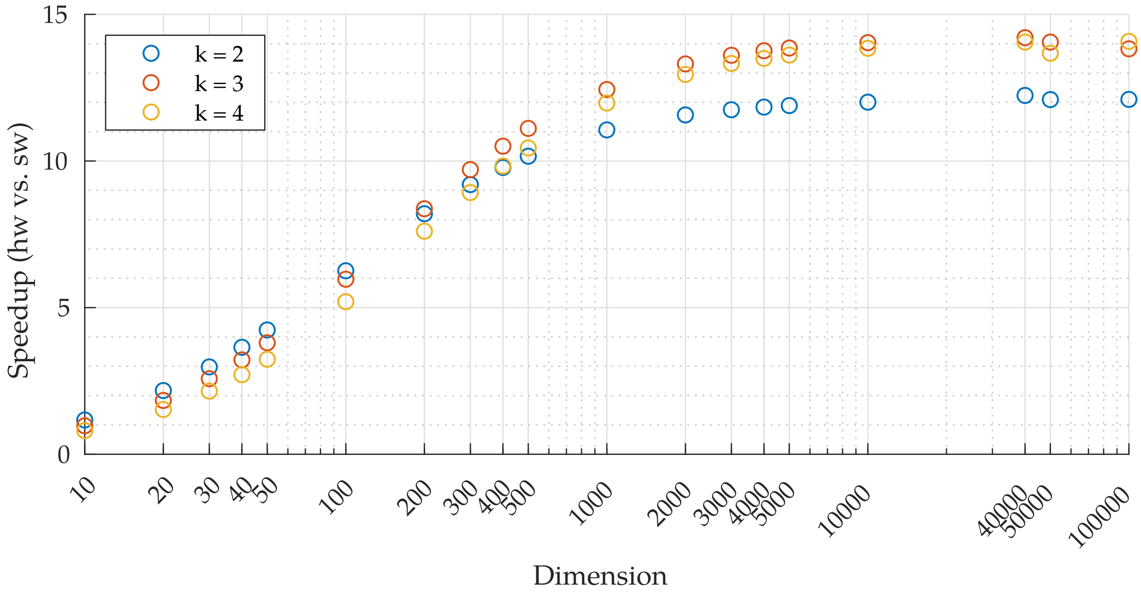


Figure 5.6: Speedup of the Gram-Schmidt process implementation using the proposed posit arithmetic hardware accelerator compared to Gram-Schmidt calculation in software for different number of vectors and vector dimensions.

where Q is an orthogonal matrix and R an upper triangular matrix. The orthogonal matrix Q in the QR-decomposition process can be computed by performing the Gram-Schmidt process on the given matrix A.

The described Gram-Schmidt process consists of multiple different vector operations such as vector subtraction, scalar multiplication and vector dot product calculations. These vector operations can be performed using the proposed posit arithmetic accelerator using the provided software library that is able to pass the correct instruction and input data to the accelerator hardware (refer to Section 5.1). Fig. 5.6 shows the performance of the described Gram-Schmidt process implementation compared to traditional execution times in software, where an emulation of the posit number format is performed. The performance of the accelerator is measured for different input vector lengths. Furthermore, the performance is measured for different number of input vectors k. From this figure we can see that the achieved speedup is dependent on the dimension of the input matrix as well as the number of input vectors. This speedup stays relatively constant for dimension values higher than 5000. Similar to the discussed evaluation of the posit dot product performance, hardware saturation occurs around this point. The main reason for the lower performance of the posit Gram-Schmidt computation compared to the accelerator implementation is that no dedicated hardware exists on a CPU to perform posit arithmetic operations. Therefore, posit arithmetic currently relies on emulation of the posit format.

5.4 Summary

In this chapter we described a novel design of a Level 1 BLAS vector arithmetic accelerator for vectors consisting of posit numbers. The accelerator is enabled with a coherent hardware-software interface using the CAPI SNAP framework. Input data is fed from column buffers represented by the Apache Arrow in-memory format, and is able to be fetched directly by the proposed accelerator. The speedup of the hardware accelerator compared to software emulation of the posit number format is dependent on the length of the input vectors. For example, for the calculation of the vector dot product for an input vector length of 10^6 elements, a speedup of approximately $15000\times$ is achieved for the machine configuration as described in Section 5.3. The achieved decimal accuracy of the posit dot product operation is on average one decimal of accuracy higher compared to posit emulation in software. Note that both software calculations of the vector dot product have been computed using a regular loop mechanism. One could improve the accuracy of these calculations by making use of special software libraries for performing BLAS operations such as the Intel Math Kernel library [41].

As described in Section 5.2, the accelerator implementation proposed in this chapter utilizes approximately 40% of the resources for the targeted FPGA. Therefore, for this platform, there are multiple ways of utilizing the remaining area available. One of the options for improving the current design is by extending the accelerator with support for multiple posit configurations. Another option is to run multiple identical accelerator cores in parallel. These cores could work on the same input vector in parallel, or work on different input vectors. In the case where the accelerator cores work on the same input vectors, the final results of the accelerator cores are to be combined in order to obtain the resulting output vector.

When comparing the proposed vector arithmetic accelerator to related work, such as the posit vector dot product accelerator presented by Chen et al. [9] (as discussed in Section 2.3), several observations can be made. The implementation presented in that work has a working frequency of $200\,\text{MHz}$, supporting vector lengths up to 1024 to 32K elements, depending on the target platform. While the working frequency of the implementation presented in our work is set at a lower $125\,\text{MHz}$, this implementation is able to continuously read posit column vector elements represented in the Apache Arrow format without a fixed limit on the maximum supported input vector length.

A practical application of the proposed vector arithmetic accelerator is shown by evaluating the Gram-Schmidt process on a set of different input matrices. For this implementation of the Gram-Schmidt application, each required vector operation is performed in a separate call to the accelerator. For every operation performed, the accelerator will load the input vectors, process the data and output the result back to the host. Naturally, this process could be significantly accelerated by removing the communication overhead between the host and the accelerator. This might be a suitable goal for future work, as this application mainly serves as a test case for evaluating the individual vector operations that the proposed accelerator is able to perform.

Based on the overall results shown for the presented posit vector arithmetic accelerator we can conclude that the application of hardware acceleration for performing posit arithmetic on (large) input vectors is beneficial when aiming towards improving the overall performance of an application working with posit numbers. The modularity of the proposed accelerator makes this design particularly useful in existing applications as the presented wrapper library serves as a drop-in replacement for existing software routines for performing vector arithmetic.

Chapter 6

Pair-HMM Posit Accelerator

In this chapter, we discuss several designs of a pair-HMM accelerator using posit arithmetic. For the implementation of the pair-HMM posit accelerator, two different designs are proposed: (1) a streaming-based accelerator connected through the Coherent Accelerator Processor Interface (CAPI) interface, and (2) an accelerator reading columnar data through the Apache Arrow in-memory format and the CAPI SNAP host-accelerator interface. For both implementations we discuss the design, implementation and results in detail. The common, top-level design of the accelerators is discussed in Section 6.1, including a description of the common components for both proposed accelerator designs. Next, the streaming-based accelerator design and its implementation and result evaluation is discussed in Section 6.2. Section 6.3 covers the pair-HMM accelerator capable of interfacing with Apache Arrow. A final summary and an overall comparison between both proposed accelerator designs is discussed in Section 6.4.

6.1 Overall Design

As discussed in Section 3.1.2, a Systolic Array (SA) architecture proves to be an efficient architecture for pair-HMM accelerators. For the proposed pair-HMM hardware accelerator with posit arithmetic we therefore implement the design using a systolic array architecture in order to parallelize the calculation of the elements in the pair-HMM matrices as described in Section 3.1.1. The architecture proposed for this implementation is based on a fixed-size systolic array design that is optimized for minimal overhead [31]. Fig. 6.1 shows an overview of the pair-HMM accelerator core design. As can be seen, the input data of the first Processing Element (PE) is fed from the pair-HMM controller. This data consists of control signals (enable/valid signals), the input test case reads and the transmission/emission probabilities corresponding to these reads. Furthermore, the initial constant used in the forward algorithm calculation (refer to Section 3.1.1) is provided.

The information received from the host, which includes the pair-HMM test cases, are being processed by the individual computing elements within the accelerator core, such as the Processing Elements (PEs) and the final score accumulator. These components are discussed in detail at the end of this chapter. An overview of the design of a PE is dis-

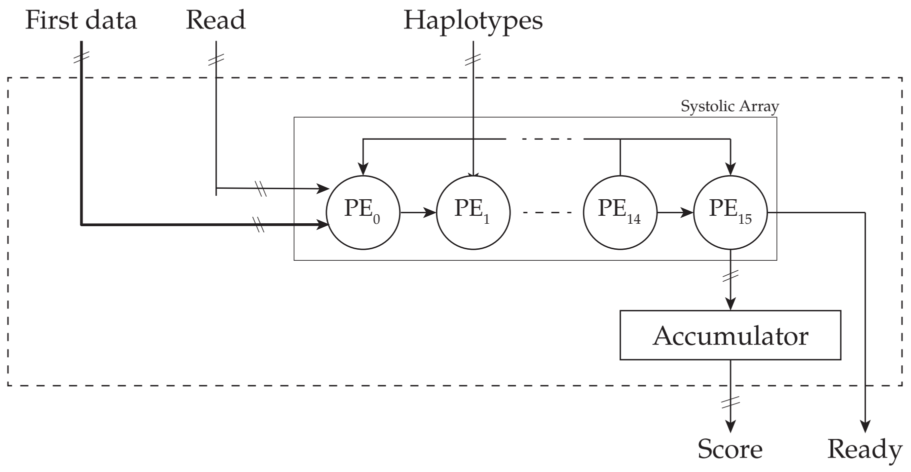


Figure 6.1: Overall overview of the pair-HMM accelerator core design.

cussed in detail in Section 6.1.1. The number of PEs in the systolic array, alternatively called the *depth*, determines the number of matrix elements, or cell updates, that can be calculated in parallel. The implemented pair-HMM accelerator core consists of 16 PEs, each calculating one element of the three pair-HMM matrices (M, I_x, I_y) as described in Section 3.1.1.

6.1.1 Processing Element (PE)

A schematic overview of the various arithmetic operations performed per PE is shown in Fig. 6.2. As each PE in the systolic array is connected in series, each PE receives calculation dependencies from the previous PE. These dependencies consist of the top-left, top and left elements of the M, I_x and I_y pair-HMM matrices (Section 3.1.1). The PE then calculates the matrix elements for the current (i, j)-position according to the inner loop of the forward algorithm as described in Section 3.1.1. Together with the previously calculated matrix elements, the corresponding emission and transmission probabilities are received from the previous PE.

The arithmetic operations performed in the PE are usually performed using floating point arithemtic. For this design, all calculations are performed using posit arithmetic. This is done by integrating the posit adder and multiplier units as described in Chapter 4. As can be seen in the schematic overview of the PE, the PE design contains both 4-cycle and 8-cycle arithmetic units in order to match the total latency of all data paths such that all newly computed matrix elements arrive at the proper clock cycle. Therefore, both 4-stage and 8-stage pipelined posit arithmetic units are implemented.

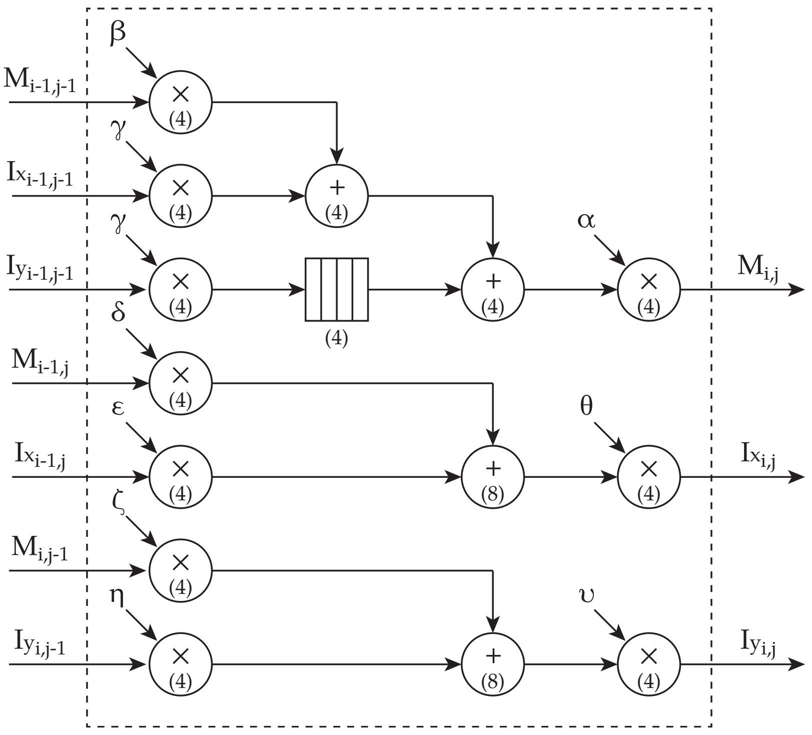


Figure 6.2: Schematic overview of a Processing Element. The latency (in clock cycles) of each unit is indicated between parentheses.

6.1.2 Accumulator

As described in Section 3.1.1, the elements of the last row in the M and I matrix are added and accumulated for each column. These matrix elements are calculated by the last PE in the systolic array design. This section describes two approaches to implementing the aforementioned accumulator.

2-Adder Accumulator

In order to implement the accumulation mechanism of the final matrix elements coming from the PEs, one could use a design that involves two adder units as depicted in Fig. 6.3. Both matrix elements calculated for the match and insertion matrices, denoted as M and I_x respectively in Section 3.1.1, serve as input to an individual posit adder. The first posit adder receives the previously accumulated result as second input, resulting in the sum of the previously accumulated result and the last element in the M matrix (refer to Section 3.1.1). This result is fed to the second posit adder, adding the last element in the I matrix. This element is delayed by the latency of the previous posit adder such that both elements are added together in the correct cycle.

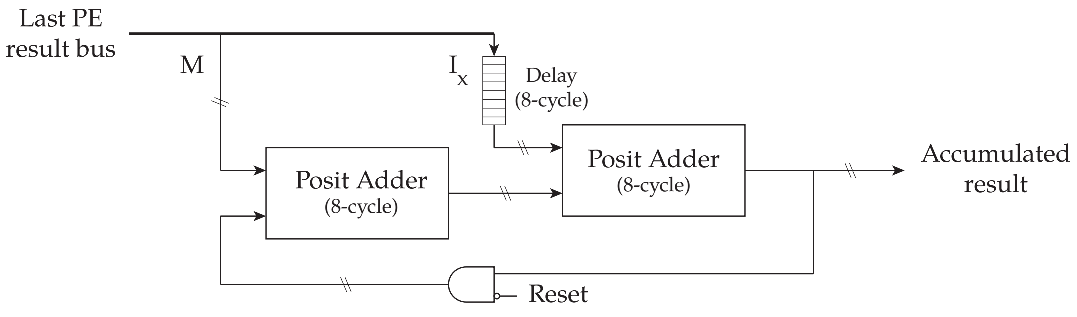


Figure 6.3: Schematic overview of the pair-HMM accumulation stage using two regular posit adder units.

As this design uses regular posit adder units, information loss can occur when adding two posit numbers due to rounding and normalization (as described in Section 4.1.2). The loss of decimal accuracy can be disadvantageous when accumulating a potentially large amount of posit numbers as the final accumulated result could have a significant error compared to its exact value when aggregating without any loss in accuracy.

Wide Accumulator

The loss of decimal accuracy induced by the use of regular posit adder units leads to the proposal of implementing the posit wide accumulator which is discussed in detail in Section 4.1.5. The posit wide accumulator can be used to implement a more accurate overall accumulator for the pair-HMM systolic array design. A schematic overview of the pair-HMM accumulator utilizing posit wide accumulator units is shown in Fig. 6.4. A regular posit adder sums the two normalized values of the accumulated pair-HMM matrix elements, resulting in the final accumulated score.

For each matrix of the pair-HMM forward algorithm (Section 3.1.1), a separate wide accumulator sums every column of its last row. The latency of a posit accumulator unit in terms of number of cycles is equal to the depth of the systolic array (16 PEs) because each matrix element is calculated per pair, thus allowing up to 16 pairs to be computed per pass through the systolic array (refer to Section 6.1). Therefore, the accumulated value for a given pair is updated every 16 cycles when new matrix elements for this pair are computed.

The advantage of this approach over the previously proposed design that uses two regular posit adders is that there is no information loss while accumulating the matrix elements of the forward algorithm. Implementing this design in the pair-HMM accelerator will result in higher area demand, however, since more logic is needed in order to process the wider fractions of accumulated values. Furthermore, this design yields a higher latency compared to the accumulator design depicted in Fig. 6.3 due to the additional final adder that is required. Therefore, the decision whether to integrate the wide accumulator design into an overall accelerator design depends on a tradeoff that is to be made between performance and desired degree of precision.

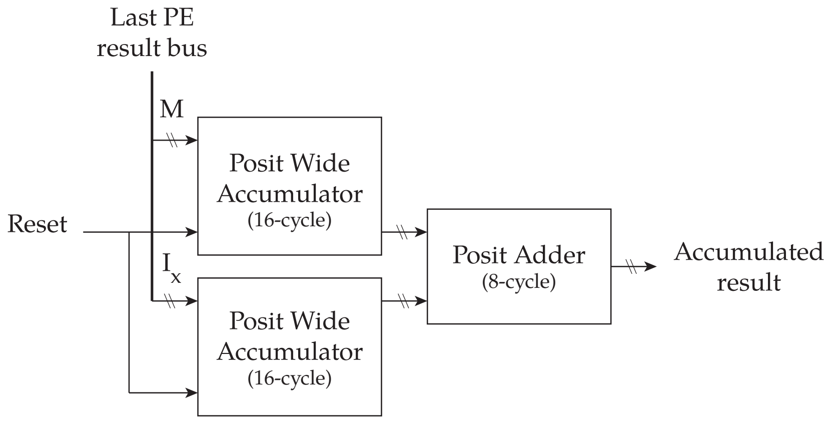


Figure 6.4: Schematic overview of the pair-HMM accumulation stage using posit wide accumulator units.

6.2 Streaming Implementation (CAPI)

In this section we will discuss a pair-HMM accelerator design with a host streaming interface using the CAPI 1.0 platform. The streaming interface is built upon the CAPI Streaming Framework [42]. First, the overall design of the accelerator will be discussed. Next, several implementations of this design are shown and analyzed based on performance with regard to decimal accuracy of the calculation results and accelerator throughput.

6.2.1 Accelerator Design

The pair-HMM accelerator receives a stream of read data (based on the input test case) from the host. The stream of base pair reads is passed to the systolic array FIFO in order to calculate the pair-HMM matrix elements. While read data is streamed through the systolic array, the corresponding haplotype base pairs that are to be compared with the read base pairs is distributed by the controller through a bus structure (as depicted in Fig. 6.1) [31].

In order to communicate with the pair-HMM accelerator and to be able to send test cases to the accelerator, a host application is implemented that is to be executed on the CPU. The host application targets the IBM POWER8 platform featuring the Coherent Accelerator Processor Interface (CAPI). This platform is widely used for setting up coherent memory interfaces between a host CPU and an external accelerator such as an FPGA. CAPI offers direct memory access (DMA) connectivity, allowing a memory region to be shared across the host and the accelerator [43]. The CAPI Streaming Framework from [42] is used to abstract away from the PSL by implementing our accelerator core in a Computing Unit (CU) interfacing with a controller and DMA engine. A schematic overview of the communication between the host and the accelerator (FPGA) is shown in Fig. 6.5.

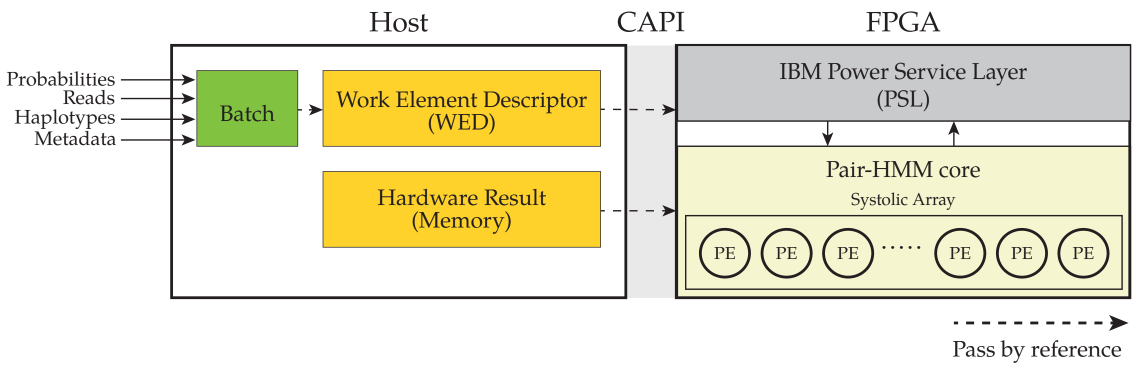


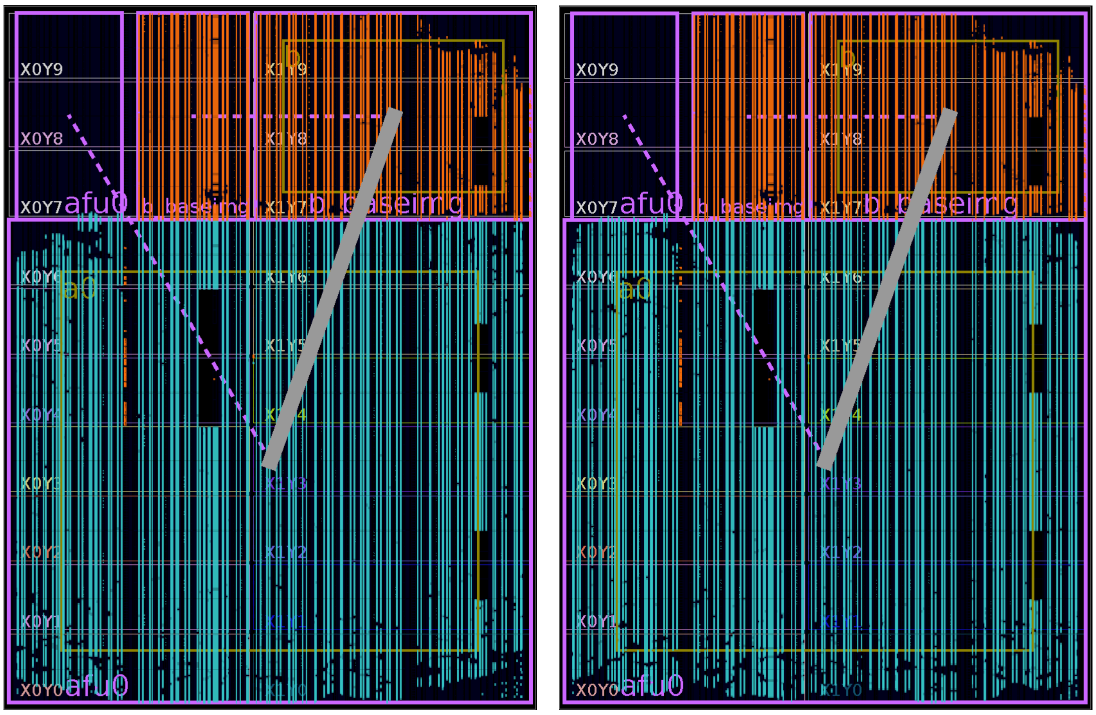
Figure 6.5: Communication between host (CPU) and FPGA through CAPI. The pair-HMM accelerator core is implemented in a Computing Unit that communicates with the Power Service Layer (PSL).

In order to provide any metadata that the pair-HMM accelerator requires about the dataset that is to be analyzed, a Work Element Descriptor (WED) is set up. The WED is a data structure containing metadata about the data that is to be processed and is located in shared memory between the host and accelerator. The pair-HMM accelerator host application fills the WED with the following information:

- Source address (memory location of the test case data)
- Destination address (memory location where the accelerator stores its results)
- Total number of workload batches

The pair-HMM host application is provided with the total number of pairs to calculate, along with the read length and haplotype sequence lengths per pair. A workload is then generated based on the provided number of pairs and read/haplotype lengths. This workload is split into multiple batches based on the number of Processing Elements in the hardware accelerator. For the systolic array design as discussed in Section 6.1 with a depth of 16 Processing Elements (PEs), the total number of batches when performing calculations for 32 pairs is equal to $\frac{\text{pairs}}{\text{\# of PEs}} = \frac{32}{16} = 2$. The number of batches is then communicated to the accelerator as part of the WED. Since batches are located adjacent to each other in memory, the FPGA is able to access these consecutive batches based on the total number of batches communicated by the WED. The host application is responsible for filling each batch with the split workload data. For each pair to be compared by the accelerator, the following data is placed in shared memory (in its corresponding batch):

- Initial value used for initialization of computation matrices
- Haplotype base pairs
- Read base pairs



(a) $\text{es} = 2$ (b) $\text{es} = 3$

Figure 6.6: Post-route layouts of the pair-HMM posit accelerator with streaming interface using the CAPI platform. FPGA device: Xilinx Virtex-7 XC7VX690T-2.

- Probabilities for each read
  - Emission probabilities (read score, θ, υ)
  - Transmission probabilities (α, β, δ, ϵ, ζ, η)

As discussed in Section 3.1.1, an initial scaling constant can be chosen that scales the first row of the initial pair-HMM matrices.

6.2.2 Hardware Implementation

The proposed pair-HMM accelerator (with streaming interface) featuring posit arithmetic is tested and analyzed for different posit configurations. As the total number of exponent bits of a posit can be configured, different implementations for the pair-HMM accelerator are possible. As described in Section 6.1.2 two ways of implementing the final score accumulator are presented. In order to optimize for calculation precision, the wide accumulator design has been implemented. Note that the Power Service Layer (PSL) runs at a frequency of $250\,\text{MHz}$. This frequency is used as a reference for generating the core frequency of the pair-HMM accelerator, which is equal to $125\,\text{MHz}$.

Configuration		Available	Used (core)		Used (total)	
`posit<32,2>`	**LUT**	433200	253891	(58.61%)	326249	(75.31%)
	Register	866400	95042	(10.97%)	173499	(20.03%)
	BRAM	1470	66	(4.49%)	374.5	(25.48%)
	DSP	3600	512	(14.22%)	536	(14.89%)
	Power		10.458 W		18.095 W	
`posit<32,3>`	**LUT**	433200	250249	(57.77%)	322662	(74.48%)
	Register	866400	97666	(11.27%)	176123	(20.33%)
	BRAM	1470	66	(4.49%)	374.5	(2.55%)
	DSP	3600	512	(14.22%)	536	(14.89%)
	Power		10.465 W		18.105 W	

Table 6.1: FPGA resource utilization and power consumption estimation of the streaming-based pair-HMM posit accelerator implementation, both for the accelerator core only and for the total implementation including the Power Service Layer.

The pair-HMM posit accelerator has been implemented and tested on an Alpha Data ADM-PCIE-7V3 FPGA accelerator card. The card features a Xilinx® Virtex-7® XC7VX690T FPGA. The post-route layouts of these implementations are depicted in Fig. 6.6. Table 6.1 shows the post-routing area usage and estimated power consumption for the target FPGA device. Note that the power consumption of the IBM PSL module (which is, among others, responsible for the interface between the host and the FPGA) is estimated at 6.975 W. The power consumption is listed for the overall design as well as for the pair-HMM accelerator core only.

6.2.3 Evaluation of Performance & Accuracy

We will evaluate the results of the pair-HMM accelerator design presented in this section. This design consists of the pair-HMM accelerator core described in Section 6.1 connected to the host through the CAPI Streaming Framework by [42] as described in Section 6.2.1.

Using the concept of the measure of decimal accuracy as defined in Section 2.6, we will analyze the precision of the posit pair-HMM accelerator. Similar to the analysis performed in Section 3.3.3, we compare the results of the posit pair-HMM accelerator design with a reference calculation performed on the host using a multi-precision arithmetic library capable of performing calculations with at least 100 decimals of accuracy [35]. Refer to Section 6.2.1 for a detailed description on the host application and how the benchmark test case is transferred to the hardware accelerator.

The test case generated for this benchmark consists of a randomized sequence of base pairs (`A`, `C`, `T`, `G`) for both the read as well as the haplotype base pair sequences. The emission and transmission probabilities per read are randomly generated numbers in the range of $[0, 1)$, and are crafted in such a way that the exact value represented by the float and posit formats that are to be benchmarked are equal. This requirement is important as we want to have a fair precision benchmark between float and posit results. If the probabilities were to be defined in the posit format, errors could arise when converting these probabilities to the float type when used in precision benchmarks. Namely, the pair-HMM calculations with `float` would already be inaccurate as the used initial probabilities hold some error.

The workload for the comparison of decimal accuracy consists of 32 pairs. A combination of different values for the read (X) and haplotype (Y) sequence lengths are tested. The initial constant used to fill computation matrices is set to 2^{10}. For an extensive analysis of the dependency of calculation results on the chosen initial scaling constant, refer to Section 3.3.3.

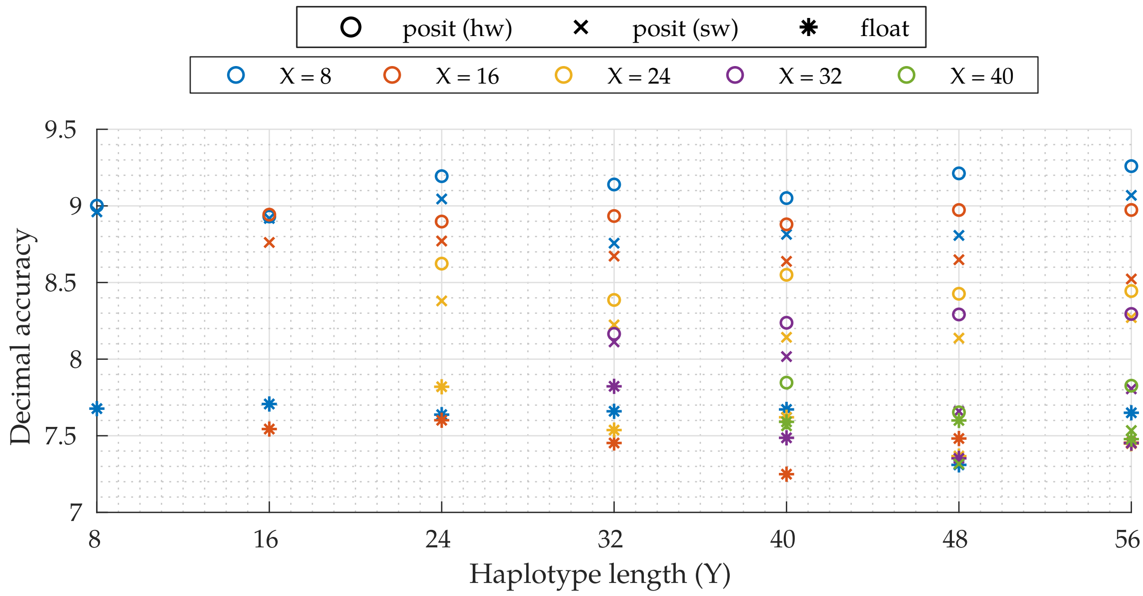
The performance in terms of throughput of the pair-HMM systolic array implementation is measured in Cell Updates Per Second (CUPS), indicating the total number of matrix elements updates inside each PE per second. Note that, as the systolic array is driven by a 125 MHz input clock, the maximum performance P_{max} in CUPS is equal to

$$P_{max} = \#\,\text{PEs} \cdot f = 16 \cdot 125 \cdot 10^6 = 2000\ \text{MCUPS} \tag{6.1}$$

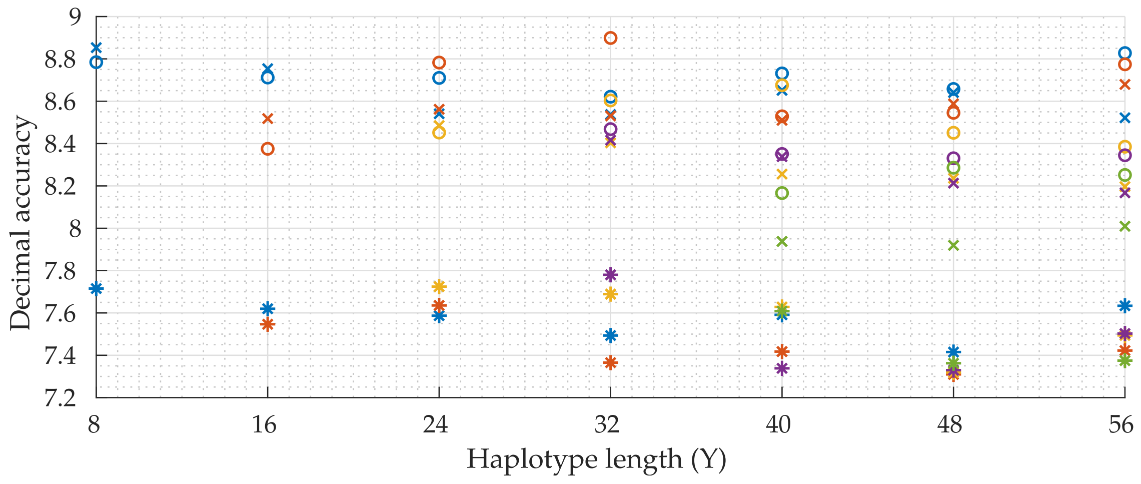
The total utilization of the systolic array is reduced for $X < \#\,\text{PEs} = X < 16$, where padding is applied for the non-utilized Processing Elements [31]. Therefore, the average number of cell updates per second will decrease for $X < 16$, as will be shown in the presented results.

Decimal Accuracy

Fig. 6.7 shows the decimal accuracy of the calculation results produced by the proposed hardware pair-HMM accelerator for different combinations of input sequence lengths X and Y. Furthermore, the decimal accuracy of the same computations performed using the `float` format are shown for comparison. The initial scaling constant (as described in Section 3.1.1) is set at 2^{10}. As can be seen, the posit number format performs better than the `float` format for all combinations of input sequence lengths X and Y. This is the case for both the `posit<32,2>` as well as the `posit<32,3>` number format. At first glance, the presented results show a superior performance of the posit number format compared to the traditional IEEE 754 float format. However, the results should be interpreted with caution. As discussed in Section 3.3.3, the decimal accuracy of the forward algorithm calculations (for all number formats) depends on the initial scaling constant chosen. Nevertheless, as was shown in Fig. 3.7, initial scaling constants exist where the maximum achievable decimal accuracy for the posit number format exceeds the maximum achievable decimal accuracy for the `float` format.



(a) `posit<32,2>`



(b) `posit<32,3>`

Figure 6.7: Decimal accuracy of the proposed pair-HMM hardware accelerator results, compared to traditional `float` computation for `posit<32,2>` and `posit<32,3>`. X and Y denote the read and haplotype input sequence lengths respectively.

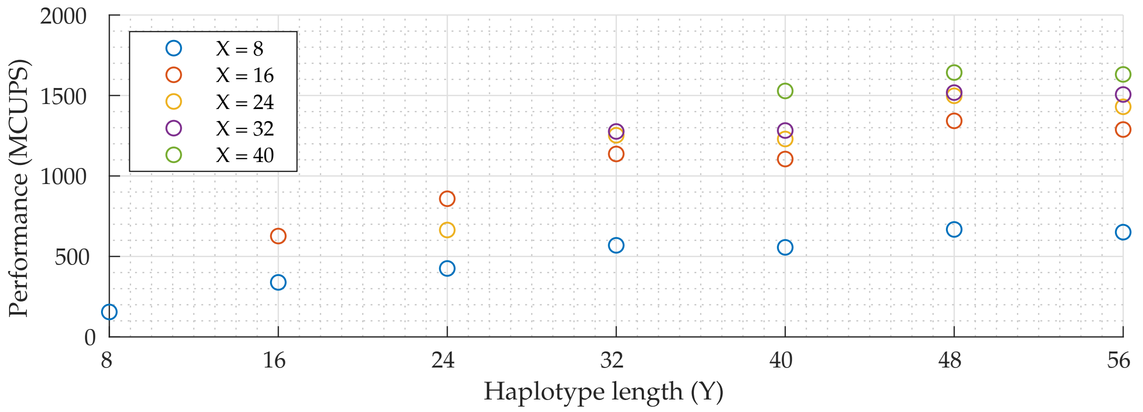


Figure 6.8: Performance in terms of throughput (in MCUPS) of the proposed pair-HMM accelerator design. X and Y denote the read and haplotype input sequence lengths respectively.

Performance

The average performance in terms of MCUPS for different combinations of sequence lengths X and Y is depicted in Fig. 6.8. This performance benchmark is performed for 10^{14} base pair comparisons. These performance measurements include the hardware overhead caused by initialization and reading and writing of the input and output data respectively. As mentioned in the introduction of this section, the average number of cell updates per second is decreased for values of X lower than the number of PEs available (16), as the systolic array becomes underutilized. This can also be seen in Fig. 6.8. The calculated theoretical maximum throughput in MCUPS (Eq. (6.1)) is not completely reached. This can be explained by the additional hardware overhead for this design, where input batch data is loaded into the accelerator buffers between consecutive batches. The performance could be improved by loading the next batch during calculation of the previous batch (as described in Section 3.1.2).

6.3 Higher-Accuracy In-Memory Hardware Implementation

In this section, we discuss the implementation of the pair-HMM accelerator that is able to read columnar data represented by the Apache Arrow columnar in-memory data format. Recall that the streaming-based accelerator discussed in Section 6.2 is designed to stream in the haplotype and read data located in host memory, making use of the CAPI Streaming Framework [42]. In this design, the input data is stored in the Apache Arrow in-memory format. The advantage of using this format is the fact that it is widely used for representing in-memory data, being language-independent and capable of storing large amounts of data. Our application, which lies in the field of genome analytics, often has to deal with large quantities of information that have to be processed. Therefore, integration with the Apache Arrow framework makes this design suitable for contemporary computing infrastructures. The Apache Arrow platform, format and its advantages and/or disadvantages are discussed in Section 2.7. Similar to the posit vector arithmetic accelerator design (discussed in Section 5.2), the CAPI SNAP framework is used in order to interface between host memory and the accelerator.

For this design, intermediate results of calculations performed inside a Processing Element (PE) as depicted in Fig. 6.2 are kept unrounded whenever possible, using the posit arithmetic framework presented in Chapter 4. The goal for this improvement is to improve the overall decimal accuracy of the final likelihood computation results produced by the pair-HMM accelerator by means of the forward algorithm.

After a detailed discussion of the accelerator design in the next section, we analyze the implementation of this design in Section 6.3.2. This implementation is then evaluated based on the decimal accuracy of the calculation results and the accelerator performance in terms of throughput.

6.3.1 Accelerator Design

Similar to the streaming-based pair-HMM accelerator, as described in Section 6.2, the input to the accelerator consists of a set of haplotype base pairs, read base pairs and the emission and transmission probabilities related to these reads. The pair-HMM algorithm is then performed on this input data using posit arithmetic. The posit arithmetic operations are designed and implemented through the arithmetic framework as described in Section 4.1.

Contrary to the streaming-based design discussed in Section 6.2.1, the data source for this implementation of the pair-HMM accelerator is provided by an in-memory representation served through the Apache Arrow framework. Therefore, the accelerator design incorporates column readers and writers in order to retrieve and store data from and to the host memory. The connection between host and accelerator is made through the CAPI SNAP framework. Similar to the vector arithmetic accelerator proposed in Section 5.2, the Fletcher framework is used [39] for retrieval of data stored in the Apache Arrow format. The Fletcher framework provides the *ColumnReader* component, which is able to read from an Arrow column. Furthermore, the *ColumnWriter* component can be utilized in order to write to Arrow columns located in the host memory.

The Arrow schema designed for this implementation is depicted in Table 6.2. As can be seen, the scheme consists of two separate tables used to represent the haplotypes as well as the reads for a specific batch. The haplotype and read base pairs are represented by an 8-bit wide field, being able to represent any ASCII character. For each read, the emission and transmission probabilities (refer to Section 3.1.1) for this read are located in the second column of this table. The probability α can contain a penalty if the read and haplotype base pairs are not equal during the pair-HMM forward algorithm evaluation (refer to Appendix A). Hence, two values for this probability are stored. As there are eight emission and transmission probabilities in total, the width of this column is equal to 256 bits, as each probability is represented by a 32-bit posit number.

Haplotypes	
	haplo (8-bit)
0	base pair
	…
	base pair
1	base pair
	…
	base pair
$\vdots$	…

Reads		
	read (8-bit)	**probabilities (256-bit)**
0	base pair	$\alpha_{\text{diff}}\ \alpha_{\text{simi}}\ \beta\ \gamma\ \delta\ \epsilon\ \eta\ \zeta$
	…	…
	base pair	$\alpha_{\text{diff}}\ \alpha_{\text{simi}}\ \beta\ \gamma\ \delta\ \epsilon\ \eta\ \zeta$
1	base pair	$\alpha_{\text{diff}}\ \alpha_{\text{simi}}\ \beta\ \gamma\ \delta\ \epsilon\ \eta\ \zeta$
	…	…
	base pair	$\alpha_{\text{diff}}\ \alpha_{\text{simi}}\ \beta\ \gamma\ \delta\ \epsilon\ \eta\ \zeta$
$\vdots$	…	…

Table 6.2: Schematic overview of the Arrow schema for the Arrow pair-HMM Accelerator implementation, consisting of the columns used to feed the pair-HMM accelerator.

The entry index indicated in the diagram represents the batch to be processed by the accelerator. The accelerator is able to access specific batches based on this index, as will be illustrated later. As the amount of base pairs inside one batch is variable, the length of each entry is also variable. When an entry is read by the accelerator, it also receives the length of this entry.

A schematic overview of the high-level components of this pair-HMM accelerator design is depicted in Fig. 6.9. For the input basepair reads, a column reader is used in order to read the base pairs and emission/transmission probabilities from the Apache Arrow data structure. A second column reader is instantiated for reading the basepairs from the input haplotype Arrow column. The data output of the column readers are fed into FIFOs. The FIFO control signals are controlled by an overall scheduler that makes sure the input data is fed into the systolic array at the correct cycle. The posit fields of the input probabilities, represented as 32-bit posit numbers, are extracted using the posit extraction unit as discussed in Section 4.1.1.

The outgoing calculation results from the systolic array, being raw posit values with unrounded fraction fields (as described in Chapter 4), are then normalized. The normalized 32-bit posit words are fed into a column writer in order to write the results into an Arrow column residing in host memory. These results are buffered by a FIFO, ready to be absorbed by the column writer.

In order to enable parallelism in the overall pair-HMM accelerator, the accelerator core is designed such that it is possible to instantiate multiple instances of the proposed pair-HMM accelerator core. A read/write bus arbiter can be used in order to connect multiple cores to the overall read/write bus of the CAPI SNAP framework. A schematic overview of this setup is depicted in Fig. 6.10. As mentioned in the description of the Arrow schema for this implementation, each batch stored in the Arrow memory format is addressable by its own index. Each accelerator core features a *batch offset* register that is writable by the host through Memory Mapped Input/Output (MMIO). As multiple batches can be

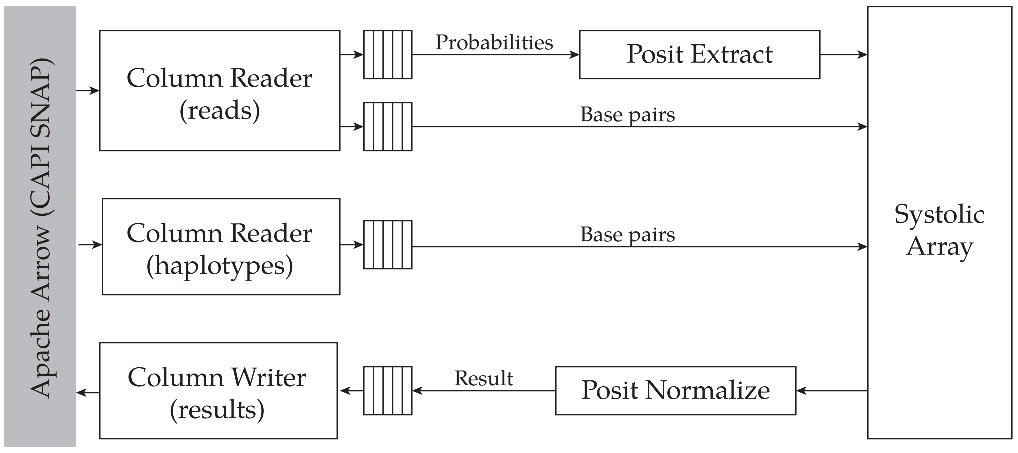


Figure 6.9: Schematic overview of the high-level components inside the pair-HMM accelerator core design, interfacing with Apache Arrow.

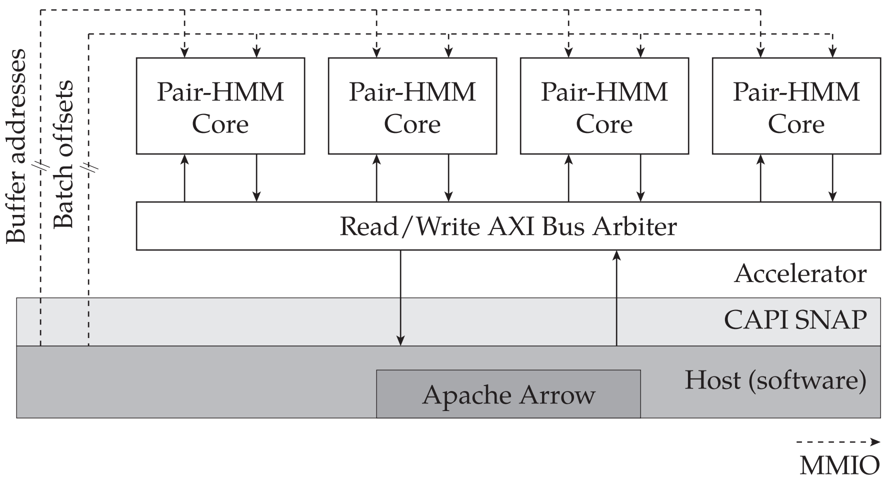
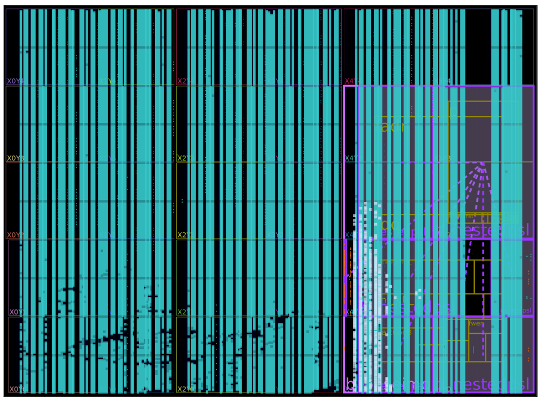


Figure 6.10: Schematic overview of the high-level connection between multiple pair-HMM accelerator core instantiations and the host, interfacing with a read/write bus arbiter connected to the CAPI SNAP AXI bus.

prepared for the columns as depicted in the Arrow schema shown in Table 6.2, each accelerator core is able to work on a different batch individually by addressing the correct batch based on the batch offset assigned to the accelerator core. While the address pointers for the Arrow input columns are equal for every instance of an accelerator core, separate buffers are instantiated for storing the results of every accelerator core. The reason for this is that it is possible for an accelerator to work on multiple batches consecutively, and therefore each accelerator is able to append the results of each batch to its own result buffer.


(a) es = 2

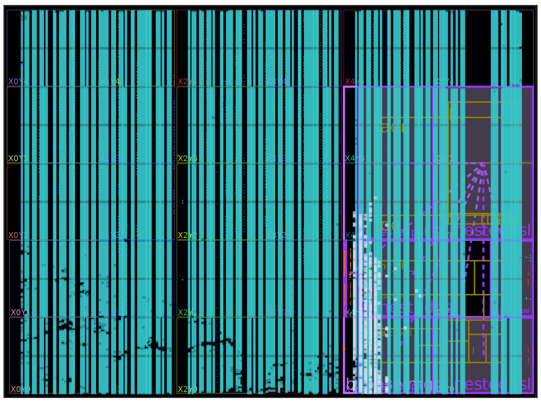

(b) es = 3

Figure 6.11: Post-route layouts of the pair-HMM posit accelerator interfacing with Apache Arrow in-memory data using the CAPI SNAP platform. FPGA device: Xilinx UltraScale XCKU060.

6.3.2 Hardware Implementation

In this section, we discuss the implementation of the pair-HMM accelerator design as discussed in Section 6.3.1. An implementation has been generated and tested for the `posit<32,2>` and `posit<32,3>` configurations. The post-route layouts of these implementations are depicted in Fig. 6.11. For both configurations, the design is implemented with a single pair-HMM accelerator core. The target FPGA for these implementations is the Xilinx Kintex® UltraScale™ XCKU060 FPGA, which is supported by the OpenPOWER CAPI SNAP framework.

Table 6.3 shows the area utilization statistics for the posit dot product accelerator implementations, along with estimated power consumptions. The power consumption for only the accelerator core as well as for the total design is displayed. The overall design includes the implementation of the Power Service Layer (PSL), required for interfacing with the host using CAPI [44].

6.3.3 Evaluation of Performance & Accuracy

Similar to the evaluations performed for the streaming-based implementation discussed in Section 6.2.3, we analyze the implementations of the accelerator design presented in this section with regard to decimal accuracy of calculation results and performance in terms of throughput as well as speedup compared to similar software implementations of the pair-HMM algorithm. The machine used in these experiment is the IBM® Power Systems™ S822LC featuring two 10-core POWER8 CPUs running at 2.92 GHz. This machine is equipped with the Alpha Data ADM-PCIE-KU3 accelerator card featuring the Xilinx Kintex® UltraScale™ XCKU060 FPGA used for this design.

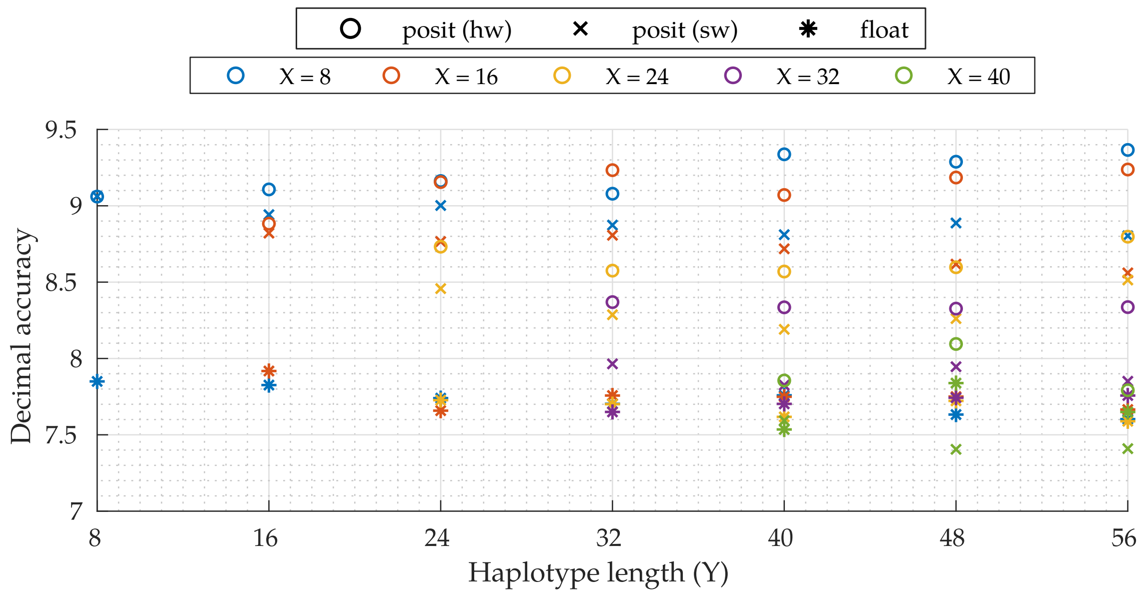
Configuration		**Available**	**Used (core)**		**Used (total)**	
`posit<32,2>`	**LUT**	331680	185174	(55.83%)	264078	(79.62%)
	Register	663360	179229	(27.02%)	271031	(40.86%)
	BRAM	1080	99	(9.17%)	425	(39.35%)
	DSP	2760	704	(25.51%)	723	(26.20%)
	Power		18.299 W		25.379 W	
`posit<32,3>`	**LUT**	331680	191827	(57.83%)	270820	(81.65%)
	Register	663360	186591	(28.13%)	278385	(41.97%)
	BRAM	1080	99	(9.17%)	425	(39.35%)
	DSP	2760	704	(25.51%)	723	(26.20%)
	Power		17.412 W		24.479 W	

Table 6.3: FPGA resource utilization and power consumption estimation of the pair-HMM posit accelerator implementation for Apache Arrow, both for the accelerator core only and for the total implementation including the Power Service Layer.

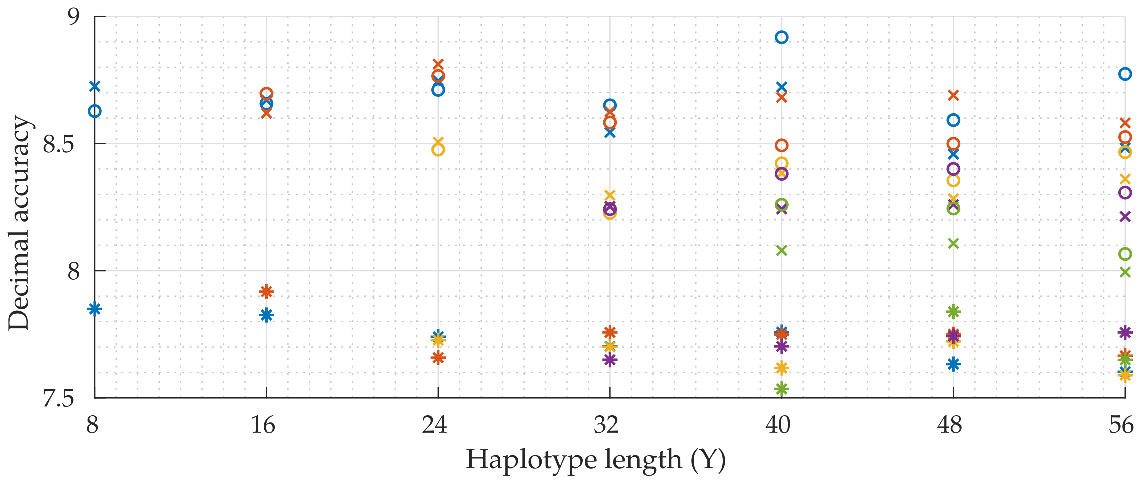
Decimal Accuracy

Fig. 6.12 shows the decimal accuracy of the calculation results produced based on simulation of the proposed hardware pair-HMM accelerator. Similar to the evaluation performed for the streaming-based accelerator described in Section 6.2.3, the decimal accuracy of the `posit<32,2>` and `posit<32,3>` hardware implementations are evaluated, together with a software evaluation of the pair-HMM forward algorithm using the `float` format. For these evaluations, different combinations of input sequence lengths X and Y have been tested. The initial scaling constant (as described in Section 3.1.1) is set at 2^{10}. For these conditions, both the software and accelerator calculation results (using both the `posit<32,2>` and `posit<32,3>` configurations) is performing better than the traditional `float` format for nearly every test case, with an increase in decimal accuracy ranging between approximately 0.5 and 2 decimals of accuracy.

As discussed in Section 6.2.3, appropriate caution should be taken with regard to the presented results. All pair-HMM forward algorithm calculations heavily depend on the initial conditions. These conditions are, apart from the input read/haplotype bases and emission/transmission probabilities, influenced by the chosen initial scaling constant. The comparison of different initial scaling constants and their effect on the decimal accuracy of final calculation results as depicted in Fig. 3.7 shows this behavior, along with the proof that scaling constants exist that result in better decimal accuracy compared to the best achievable decimal accuracy for the `float` format.

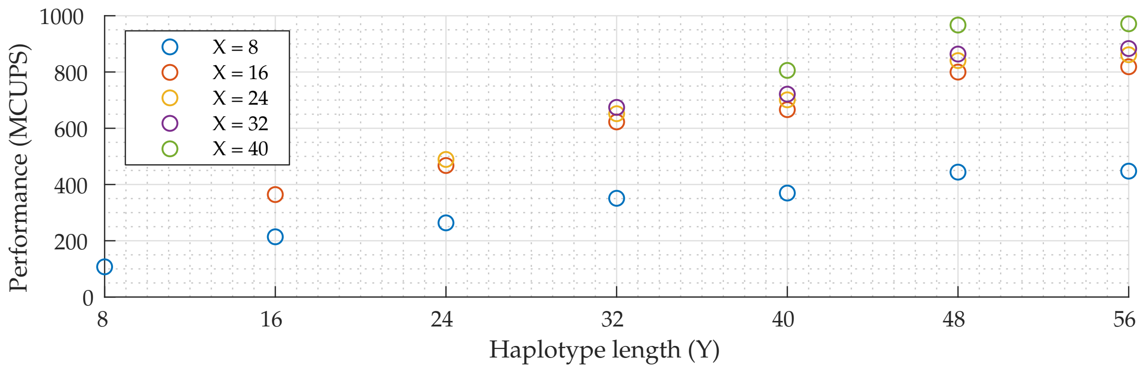


(a) `posit<32,2>`

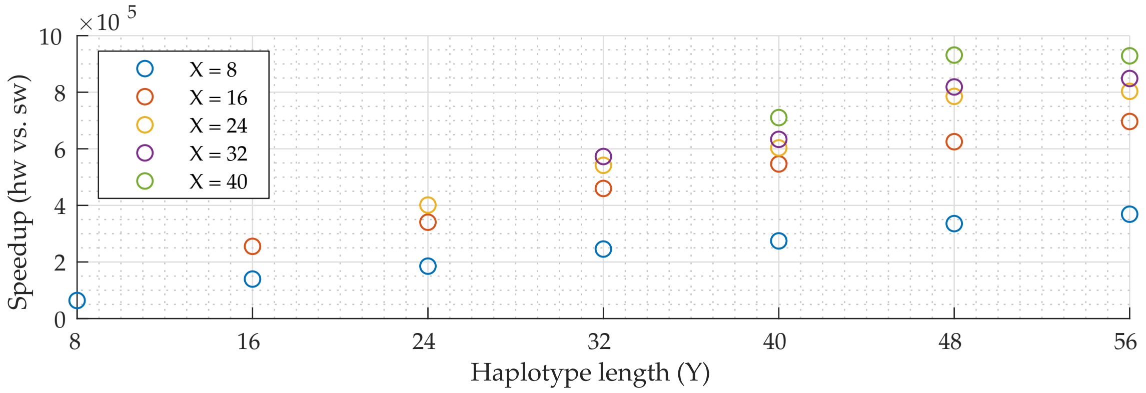


(b) `posit<32,3>`

Figure 6.12: Decimal accuracy of the proposed pair-HMM hardware accelerator results, compared to traditional `float` computation for `posit<32,2>` and `posit<32,3>`. X and Y denote the read and haplotype input sequence lengths respectively.



(a) Performance in terms of throughput (in MCUPS).



(b) Speedup of hardware versus software, based on total execution time.

Figure 6.13: Performance in terms of throughput (in MCUPS) and speedup compared to software calculation for the proposed pair-HMM accelerator design. X and Y denote the read and haplotype input sequence lengths respectively.

Performance

Similar to the analysis of the streaming-based pair-HMM accelerator discussed in Section 6.2.3, the performance in terms of throughput is measured in CUPS which indicates the total number of matrix elements updates performed by each PE per second. The maximum theoretical throughput P_{max} is equal to 2000 MCUPS as determined by Eq. (6.1).

The average performance for the pair-HMM hardware accelerator interfacing with the Apache Arrow columnar memory format implementation in terms of MCUPS for different combinations of sequence lengths X and Y is depicted in Fig. 6.13a. Similar to the evaluation performed for the streaming-based accelerator, this performance benchmark is performed for 2^{15} base pair comparisons. As can be seen, the throughput decreases for any input sequence length X lower than the number of PEs in the systolic array due to underutilization of the overall accelerator. The theoretical maximum throughput of 2000 MCUPS is not fully reached due to the present hardware overhead. The explanation for this is similar to the reason stated during the evaluation of the performance of the streaming-based accelerator (Section 6.2.3): batch data is loaded into the accelerator buffers between batches, and the next batch will be loaded after finishing the previous batch. The overhead between initiating the read request to the host and receiving the full data set decreases the maximum achievable performance. The speedup of the pair-HMM hardware accelerator calculations compared to calculation in software (using a posit format emulation library) is depicted in Fig. 6.13b for the same data sets. A significant speedup is observed for all tested combinations of read and haplotype input sequence lengths, ranging from a factor of approximately 10^5 to 10^6 times speedup.

6.4 Summary

In this chapter, two approaches are presented to implementing a hardware accelerator for the pair-HMM forward algorithm using posit arithmetic. The first design that was described and implemented is an accelerator based on a streaming interface connected with the CAPI 1.0 platform (Section 6.2). The second proposed design uses the CAPI SNAP framework and is able to interface with in-memory data represented in the Apache Arrow columnar memory format, and was discussed in Section 6.3. Furthermore, this implementation consists of posit arithmetic units that are able to perform calculations without intermediate rounding during the calculation of cell updates of the pair-HMM matrix (as described in Section 3.1.1).

Decimal Accuracy

The decimal accuracy of calculation results for both pair-HMM accelerator designs are evaluated. On average, the streaming-based pair-HMM algorithm achieves an increase of approximately 0.5 decimals of accuracy compared to software posit emulation for the same input data set, consisting of a read sequence length of 40 base pairs and a haplotype sequence length of 56 base pairs. Overall, the posit number format beats the IEEE floating point number format in terms of decimal accuracy. Note, however, that the presented decimal accuracy measurements have been performed for a specific initial scaling constant, which partly determines the accuracy of calculation results as described in Section 3.3.3. However, as discussed in the same section, initial scaling constants exist that outperform the maximum achievable decimal accuracy for the IEEE float format.

The second proposed accelerator implementation (discussed in Section 6.3), interfacing with the Apache Arrow columnar memory format, achieves similar results in terms of decimal accuracy of the final calculation results when comparing with the previously introduced implementation. Although this implementation consists of posit arithmetic units with a wider range of fraction bits available, a significant improvement in decimal accuracy is not observed. Since only the arithmetic units inside the processing elements of the systolic array are equipped with the posit arithmetic units that accept and produce unrounded intermediate results (Chapter 4), the in- and output of each PE is normalized. The main reason for this trade-off is the limited area available for the targeted hardware platform.

Performance

The measured throughput in terms of Cell Updates Per Second (CUPS) is equal to approximately 1600 MCUPS for the streaming-based accelerator and 1000 MCUPS for the second implementation, interfacing with the Apache Arrow columnar memory format. The reason for the lower performance of the second implementation is similar to the reason stated for the streaming-based accelerator. During different calculation batches, the accelerator loads the next batch that is located inside the Apache Arrow data format. The overhead associated with loading the the input data into the accelerator buffers therefore becomes significant for small input batches. Improvements to this mechanism can be made by loading the next batch while processing the previous batch (as described in Section 3.1.2) so that the next batch computation can immediately be started.

Chapter 7

Conclusions

A new generation of number representation systems naturally draws the attention of many due to its impact on the broad field of computing. The IEEE 754 floating point standard has been the de facto standard for representing floating point numbers since hardware supported floating point arithmetic was introduced. Hence, the proposal of the novel posit number format is closely monitored and evaluated as it presents itself as a direct competitor to the well-established and trusted IEEE standard.

In this thesis, we investigated the capabilities of posit arithmetic to IEEE floating point arithmetic to identify their advantages and disadvantages. We will summarize the contributions and results of the work presented in this thesis. Furthermore, additional discussion and recommendations for future work are given.

Contributions & Results

In this work, the performance of the posit number format in terms of decimal accuracy has been analyzed and compared with alternative number representations.

In particular, a study of the application of posit arithmetic in the field of bioinformatics was performed. The effect on decimal accuracy of the pair-HMM forward algorithm caused by the replacement of traditional IEEE 754 floating point arithmetic by posit arithmetic is analyzed. Based on this analysis, it can be concluded that the posit number format does perform better than the traditional IEEE 754 floating point standard in terms of decimal accuracy of calculation results. Although the overall accuracy is dependent on the test case, it is proven that the best achievable decimal accuracy using posit arithmetic is higher compared to the IEEE floating point format. Combining this observation with the fact that the posit configurations that have been analyzed in this work cost the same number of bits as the float format that it is compared to, there is reason to believe that the posit number format could be a worthy successor to the IEEE 754 floating point format for evaluating the pair-HMM model.

After theoretical and empirical analyses of the application of the posit number format in existing applications, multiple designs for accelerating posit arithmetic in hardware have been proposed and implemented.

A framework for performing high-precision posit arithmetic in reconfigurable logic is presented. The supported arithmetic operations, which consist of posit vector addition/subtraction, multiplication and accumulation operations, can be performed without rounding off intermediate results. Final posit normalization takes place only after the last calculation has been performed in a computation procedure, and a rounding scheme is applied to perform correct rounding based on the truncated fraction bits. Because of these two measures, loss of decimal accuracy is minimized. The proposed posit arithmetic units achieve approximately 250 MPOPS for addition, 160 MPOPS for multiplication and 180 MPOPS for accumulation operations.

Building upon the presented posit hardware arithmetic framework, the design of an accelerator for performing vector arithmetic on posit column vectors is presented and implemented. The accelerator inhibits a modular design which enables one to develop a hybrid design where accelerated operations on (sparse) input posit vectors can be performed. The accelerator calculation results are directly useable in existing software applications using the corresponding software interface library. The performance of the hardware accelerator is dependent on the length of the input vectors. For the calculation of the vector dot product for an input vector length of 10^6 elements a speedup of approximately $15000\times$ is achieved. The decimal accuracy of the dot product results produced by the proposed accelerator is improved by one decimal of accuracy on average compared to a software implementation, which in turn yields one extra decimal of accuracy compared to calculation with the IEEE 754 floating point format. Therefore, it can be concluded that the posit number format overall performs better for the test cases presented in this work in terms of accuracy.

Based on the performed theoretical analysis of the feasibility of applying posit arithmetic to the pair-HMM forward algorithm, a hardware accelerator for the pair-HMM forward algorithm using posit arithmetic is proposed. Two versions of the hardware accelerator are implemented. The first design uses a streaming interface to connect with the host through the CAPI platform. The second design interfaces with data represented in the Apache Arrow columnar memory format using the CAPI SNAP framework. Furthermore, this design is implemented using posit arithmetic units that only perform normalization at the output of a processing element, which implements the calculation of the inner loop inside the pair-HMM forward algorithm. Overall, the posit number format beats the IEEE floating point number format in terms of decimal accuracy, ranging from an improvement of 0.5 to 1 additional decimal of accuracy. As concluded during the empirical analysis of the pair-HMM forward algorithm, the choice of initial scaling constant partly determines the achieved decimal accuracy. However, the maximum achievable decimal accuracy is higher for the posit number format compared to the IEEE float format. Similar results in terms of decimal accuracy are observed for the second implementation. In this case, enlarging the fraction fields of intermediate results during calculation of the inner loop of the pair-HMM forward algorithm does not significantly improve overall accuracy due to the normalization of intermediate results between separate processing elements in the systolic array.

The proposed accelerator has a throughput of 1600 MCUPS for the streaming-based accelerator and 1000 MCUPS for the accelerator with interface to the Apache Arrow columnar data format. The lower throughput is explained by the fact that hardware overhead influences the throughput significantly, as new batch information is loaded from the host memory between consecutive batches. This can be improved by loading the next batch already during the processing of the previous batch, and is considered future work.

Discussion & Recommendations

While preserving bits in intermediate computations could improve the decimal accuracy of the final computation result, an increase in chip area usage is naturally inevitable. Larger bus widths are required in accuracy-optimized implementations, as opposed to implementations consisting of standalone posit arithmetic units where the result is directly packed into a regular posit number. Therefore, more registers are needed and an increase in wiring is expected. This illustrates a direct trade-off between the desired decimal accuracy of computation results and the amount of chip area available. It might be of interest to analyze the point where the potential increase in decimal accuracy does not outweigh the required increase in area any more. In any case, these boundaries highly depend on the precision requirements for the application one is interested in augmenting with posit arithmetic. Furthermore, this trade off highly depends on the targeted platform. For example, for a reconfigurable target such as an FPGA the chosen platform size determines how much area is available for additional wiring, enabling more intermediate bits to be communicated between arithmetic units.

One of the key features of the posit number representation system is the configurable number of total and exponent bits. However, the fixed nature of silicon hinders the possibility of truly flexible arithmetic, where one is able to modify its posit configuration on-the-fly during computation. Therefore, posit arithmetic hardware implementations are mostly bound to a specific configuration of total and exponent bits. Multiple posit arithmetic units for a specific set of posit configurations could be implemented on a single chip, requiring additional conversion logic in order to interchange posit values between different posit configurations.

Appendix A

Pair-HMM Pseudocode

Listing A.1: Pseudocode of the pair-HMM forward algorithm as discussed in Section 3.1.1, used in the precision analysis performed in Section 3.3.3.

```

function randomNumber
   float floatNumber
   posit positNumber

   do
      floatNumber = random()
      positNumber = floatNumber
   while (positNumber != floatNumber)

   return positNumber
 end

P = 0
for $r \le$ rows
   $\alpha, \beta, \gamma, \delta, \epsilon, \eta, \zeta \leftarrow$ random number

   for $c \le$ columns
      if read base pair == haplotype base pair
         $\alpha \leftarrow 1 - \alpha$
      else
         Apply a penalty for a read base pair that is not matching with the haplotype base pair
         $\alpha \leftarrow \alpha/3$
      end

      Below variables represent M, $I_x$, $I_y$ elements for the current row and column
      $M_{r,c} \leftarrow \alpha \times (\beta \times M_{r-1,c-1} + \gamma \times I_{x_{r-1,c-1}} + \gamma \times I_{y_{r-1,c-1}})$
      $I_{x_{r,c}} \leftarrow \delta \times M_{r-1,c} + \epsilon \times I_{x_{r-1,c}}$
      $I_{y_{r,c}} \leftarrow \eta \times M_{r,c-1} + \zeta \times I_{y_{r,c-1}}$

      Accumulation of results resulting in the overall likelihood of sequences $\mathbf{x}$ and $\mathbf{y}$ being related (refer to Eq. (3.4))
      if $r$ == rows
         $P = P + M_{r,c} + I_{x_{r,c}} + I_{y_{r,c}}$
      end
   end
end
```

Appendix B

Posit Field Widths for Arithmetic Framework

w_x	bit field width (in bits) of field x
nbits	total number of bits in posit configuration
es	number of exponent bits in posit configuration

B.1 Regular Value

Sign, Infinite, Zero

$$w_{\mathrm{sign}} = 1 \qquad w_{\mathrm{infinite}} = 1 \qquad w_{\mathrm{zero}} = 1$$

Fraction

$$w_{\mathrm{fraction}} = \mathrm{nbits} - \mathrm{es} - 3 \tag{B.1}$$

Scale

$$\begin{aligned}
w_{\mathrm{scale}} &= \left\lceil \log_2 \left(\text{useed}^{\text{maximum regime}} \times \text{maximum exponent} \right) \right\rceil \\
&= \left\lceil \log_2 \left(\left(2^{2^{\mathrm{es}}}\right)^{\text{maximum regime}} \times \left(2^{\mathrm{es}} - 1\right) \right) \right\rceil \\
&= \left\lceil \log_2 \left(\left(2^{2^{\mathrm{es}}}\right)^{\mathrm{nbits}-2} \times \left(2^{\mathrm{es}} - 1\right) \right) \right\rceil \\
&= \left\lceil \log_2 \left(2^{2^{\mathrm{es}} \times (\mathrm{nbits}-2) + 2^{\mathrm{es}} - 1} \right) \right\rceil \\
&= \left\lceil \log_2 \left(2^{2^{\mathrm{es}} \times (\mathrm{nbits}-1) - 1} \right) \right\rceil \\
&= \left\lceil \log_2 \left(2^{\mathrm{es}} \times (\mathrm{nbits} - 1) - 1 \right) \right\rceil
\end{aligned} \tag{B.2}$$

B.2 Sum

Sign, Infinite, Zero

$$w_{\text{sign}} = 1 \qquad w_{\text{infinite}} = 1 \qquad w_{\text{zero}} = 1$$

Fraction

$$\begin{aligned} w_{\text{fraction}} &= w_{\text{fraction}|\text{regular}} + 1 \\ &= \text{nbits} - \text{es} - 3 + 1 \\ &= \text{nbits} - \text{es} - 2 \end{aligned} \tag{B.3}$$

Scale As the lowest input operand is matched by scale of the larger input operand, the maximum scale does not change, and hence is equal to the maximum scale value of a regular posit value.

$$\begin{aligned} w_{\text{scale}} &= w_{\text{scale}|\text{regular}} \\ &= \lceil \log_2 (2^{\text{es}} \times (\text{nbits} - 1) - 1) \rceil \end{aligned} \tag{B.4}$$

B.3 Product

Sign, Infinite, Zero

$$w_{\text{sign}} = 1 \qquad w_{\text{infinite}} = 1 \qquad w_{\text{zero}} = 1$$

Fraction

$$\begin{aligned} w_{\text{fraction}} &= w_{\text{fraction}|\text{regular}} \times 2 \\ &= 2 \times (\text{nbits} - \text{es} - 3) \end{aligned} \tag{B.5}$$

Scale For multiplication of two input operands, the product scale is equal to the sum of both input operand scales. Hence, the number of bits required to represent the scale is increased by 1.

$$\begin{aligned} w_{\text{scale}} &= w_{\text{scale}|\text{regular}} + 1 \\ &= \lceil \log_2 (2^{\text{es}} \times (\text{nbits} - 1) - 1) \rceil + 1 \end{aligned} \tag{B.6}$$

B.4 Accumulation

Sign, Infinite, Zero

$$w_{\text{sign}} = 1 \qquad w_{\text{infinite}} = 1 \qquad w_{\text{zero}} = 1$$

Fraction The width of the fraction field should be equal to the fraction size of a regular posit value plus the maximum amount an input fraction might be shifted in order to match the scale of both input operands. This shift is largest when the scale difference between two input operands is at a maximum, which is equal to the maximum regime scale value $(2^{\mathrm{es}} \times (\mathrm{nbits} - 2))$.

$$\begin{aligned} w_{\mathrm{fraction}} &= w_{\mathrm{fraction}|\mathrm{regular}} + \text{maximum shift} \\ &= \mathrm{nbits} - \mathrm{es} - 3 + 2^{\mathrm{es}} \times (\mathrm{nbits} - 2) \end{aligned} \tag{B.7}$$

Scale As the lowest input operand is matched by scale of the current accumulated value, the maximum scale does not change, and hence is equal to the maximum scale value of a regular posit value.

$$w_{\mathrm{scale}} = w_{\mathrm{scale}|\mathrm{regular}} \tag{B.8}$$

$$= \lceil \log_2 (2^{\mathrm{es}} \times (\mathrm{nbits} - 1) - 1) \rceil \tag{B.9}$$

Bibliography

[1] J. L. Gustafson, *The End of Error: Unum Computing*. CRC Press, Feb. 2015.

[2] J. L. Gustafson and I. T. Yonemoto, "Beating Floating Point at its Own Game: Posit Arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, Apr. 2017. [Online]. Available: `http://superfri.org/superfri/article/view/137`.

[3] J. L. Gustafson, "A Radical Approach to Computation with Real Numbers," *Supercomputing Frontiers and Innovations*, vol. 3, no. 2, pp. 38–53, Jul. 2016.

[4] "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Std 754-1985*, 1985.

[5] W. Kahan, *Prof. W. Kahan's Commentary on "THE END of ERROR — Unum Computing" by John L. Gustafson*, 2015.

[6] W. Kahan, "A Critique of John L. Gustafson's THE END of ERROR — Unum Computation and his A Radical Approach to Computation with Real Numbers," Mar. 2016.

[7] J. L. Gustafson, *Posit Arithmetic*. Oct. 2017. [Online]. Available: `https://posithub.org/docs/Posits4.pdf` (visited on Feb. 3, 2018).

[8] R. Sayre, *Mozilla's New JavaScript Value Representation*, Aug. 2010. [Online]. Available: `http://tomschuster.name/sayrer-fatval-backup/cache.aspx.htm#post-485` (visited on Mar. 22, 2018).

[9] J. Chen, Z. Al-Ars, and H. P. Hofstee, "A Matrix-multiply Unit for Posits in Reconfigurable Logic Leveraging (Open)CAPI," in *Proceedings of the Conference for Next Generation Arithmetic*, Singapore: ACM, 2018, 1:1–1:5.

[10] M. K. Jaiswal and H. K. H So, "Universal Number Posit Arithmetic Generator on FPGA," *DATE18*,

[11] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture," Sep. 2016.

[12] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug. 2008.

[13] M. Cowlishaw, "Densely packed decimal encoding," *IEE Proceedings - Computers and Digital Techniques*, vol. 149, no. 3, pp. 102–104, May 2002.

[14] B. Schäling, *The Boost C++ Libraries*, 2nd ed. XML Press, 2014.

[15] Free Software Foundation, *Using the GNU Compiler Collection (GCC): Optimize Options*. [Online]. Available: `https://gcc.gnu.org/onlinedocs/gcc-8.2.0/gcc/Optimize-Options.html` (visited on Aug. 27, 2018).

[16] Intel Corporation, *Setting the FTZ and DAZ Flags — User and Reference Guide for the Intel® C++ Compiler 15.0*, Aug. 2015. [Online]. Available: `https://software.intel.com/en-us/node/523328` (visited on Jan. 24, 2018).

[17] J. Demmel, "Underflow and the Reliability of Numerical Software," *SIAM Journal on Scientific and Statistical Computing*, vol. 5, no. 4, pp. 887–919, Dec. 1984. [Online]. Available: `https://epubs.siam.org/doi/10.1137/0905062` (visited on Aug. 24, 2018).

[18] Sun Microsystems, *IEEE Arithmetic Model - Numerical Computation Guide*, 2002. [Online]. Available: `https://docs.oracle.com/cd/E19957-01/816-2464/ncg_math.html` (visited on Aug. 27, 2018).

[19] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 59–70, Jan. 1990.

[20] The Apache Software Foundation, *Apache Arrow^{TM}*. [Online]. Available: `https://arrow.apache.org`.

[21] L. E. Baum and T. Petrie, "Statistical Inference for Probabilistic Functions of Finite State Markov Chains," *The Annals of Mathematical Statistics*, vol. 37, no. 6, pp. 1554–1563, 1966. [Online]. Available: `http://www.jstor.org/stable/2238772`.

[22] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, Feb. 1989.

[23] B.-J. Yoon, "Hidden Markov Models and their Applications in Biological Sequence Analysis," *Current Genomics*, vol. 10, no. 6, pp. 402–415, Sep. 2009. [Online]. Available: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2766791/`.

[24] Broad Institute, *GATK — Doc #11078 — Evaluating the evidence for haplotypes and variant alleles (HaplotypeCaller & Mutect2)*. [Online]. Available: `https://software.broadinstitute.org/gatk/documentation/article?id=11078` (visited on Aug. 5, 2018).

[25] V. Bafna and S. C. Sahinalp, *Research in Computational Molecular Biology: 15th Annual International Conference, RECOMB 2011, Vancouver, BC, Canada, March 28-31, 2011. Proceedings*. Springer, Mar. 2011.

[26] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo, "The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data," *Genome Research*, vol. 20, no. 9, pp. 1297–1303, Sep. 2010. [Online]. Available: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2928508/`.

[27] M. Stamp, *A Revealing Introduction to Hidden Markov Models*, Jan. 2018. [Online]. Available: `http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf` (visited on Aug. 28, 2018).

[28] S. Ren, K. Bertels, and Z. Al-Ars, "GPU-Accelerated GATK HaplotypeCaller with Load-Balanced Multi-Process Optimization," in *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)*, Oct. 2017, pp. 497–502.

[29] J. Wang, X. Xie, and J. Cong, "Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 72–81.

[30] C. Rauer, G. S. Powley, M. Ahsan, and N. Finamore, *Accelerating Genomics Research with OpenCL^{TM} and FPGAs*, 2017. [Online]. Available: `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/genomics-research-with-opencl-and-fpgas-paper.pdf`.

[31] J. Peltenburg, S. Ren, and Z. Al-Ars, "Maximizing systolic array efficiency to accelerate the PairHMM Forward Algorithm," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, Dec. 2016, pp. 758–762.

[32] M. Ito and M. Ohara, "A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for pair-HMM algorithm," in *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*, Apr. 2016, pp. 1–3.

[33] G. A. Van der Auwera, M. O. Carneiro, *et al.*, "From FastQ data to high confidence variant calls: The Genome Analysis Toolkit best practices pipeline," *Current Protocols in Bioinformatics*, vol. 11, no. 1110, pp. 11.10.1–11.10.33, Oct. 2013. [Online]. Available: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4243306/`.

[34] Stillwater Supercomputing, Inc., *Universal: Universal Number Arithmetic*, Revision `67fbcaf`, 2017. [Online]. Available: `https://github.com/stillwater-sc/universal` (visited on Mar. 21, 2018).

[35] *Cpp_dec_float - 1.63.0*. [Online]. Available: `http://www.boost.org/doc/libs/1_63_0/libs/multiprecision/doc/html/boost_multiprecision/tut/floats/cpp_dec_float.html` (visited on Mar. 20, 2018).

[36] *Phred-scaled Quality Scores*. [Online]. Available: `https://gatkforums.broadinstitute.org/gatk/discussion/4260/phred-scaled-quality-scores` (visited on Feb. 21, 2018).

[37] L. van Dam, *pairhmm_posit_cpp*, Feb. 2018. [Online]. Available: `https://github.com/lvandam/pairhmm_posit_cpp`.

[38] J. A. Reuter, D. Spacek, and M. P. Snyder, "High-Throughput Sequencing Technologies," *Molecular Cell*, vol. 58, no. 4, pp. 586–597, May 2015.

[39] J. Peltenburg, *Fletcher: A framework to integrate FPGA accelerators with Apache Arrow*, 2018. [Online]. Available: `https://github.com/johanpel/fletcher` (visited on Aug. 7, 2018).

[40] "Orthogonalization," in *Encyclopaedia of Mathematics*, ser. Encyclopaedia of Mathematics, Springer Netherlands, 1994. [Online]. Available: `//www.springer.com/us/book/9781556080104` (visited on Sep. 4, 2018).

[41] Intel Corporation, *Intel® Math Kernel Library (Intel® MKL)*. [Online]. Available: `https://software.intel.com/en-us/mkl`.

[42] M. Brobbel, *Capi-streaming-framework: AFU framework for streaming applications with CAPI*, Oct. 2017. [Online]. Available: `https://github.com/mbrobbel/capi-streaming-framework` (visited on Apr. 23, 2018).

[43] B. Wile, *Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems*, Sep. 2014.

[44] OpenPOWER, *CAPI SNAP Framework Hardware and Software*, 2016. [Online]. Available: `https://github.com/open-power/snap` (visited on Aug. 2, 2018).