An Accelerator for Posit Arithmetic Targeting Posit Level 1 BLAS Routines and Pair-HMM

van Dam, Laurens ; Peltenburg, Johan; Al-Ars, Zaid; Hofstee, H. Peter

DOI
10.1145/3316279.3316284

Publication date
2019

Document Version
Final published version

Published in
CoNGA'19 Proceedings of the Conference for Next Generation Arithmetic 2019

An Accelerator for Posit Arithmetic Targeting Posit Level 1 BLAS Routines and Pair-HMM

Laurens van Dam
Delft University of Technology
Delft, Netherlands
laurens@vand.am

Johan Peltenburg
Delft University of Technology
Delft, Netherlands
j.w.peltenburg@tudelft.nl

Zaid Al-Ars
Delft University of Technology
Delft, Netherlands
z.al-ars@tudelft.nl

H. Peter Hofstee
IBM Austin Research Laboratory
Austin, TX, USA
hofstee@us.ibm.com

ABSTRACT

The newly proposed posit number format uses a significantly different approach to represent floating point numbers. This paper introduces a framework for posit arithmetic in reconfigurable logic that maintains full precision in intermediate results. We present the design and implementation of a L1 BLAS arithmetic accelerator on posit vectors leveraging Apache Arrow. For a vector dot product with an input vector length of $\mathbf{10^6}$ elements, a hardware speedup of approximately $\mathbf{10^4}$ is achieved as compared to posit software emulation. For 32-bit numbers, the decimal accuracy of the posit dot product results improve by one decimal of accuracy on average compared to a software implementation, and two extra decimals compared to the IEEE754 format. We also present a posit-based implementation of pair-HMM. In this case, the hardware speedup vs. a posit-based software implementation ranges from $\mathbf{10^5}$ to $\mathbf{10^6}$. With appropriate initial scaling constants, accuracy improves on an implementation based on IEEE 754.

KEYWORDS

posit, unum, unum-III, BLAS, arithmetic, pair-HMM, decimal accuracy, FPGA, accelerator

ACM Reference format:
Laurens van Dam, Johan Peltenburg, Zaid Al-Ars, and H. Peter Hofstee. 2019. An Accelerator for Posit Arithmetic Targeting Posit Level 1 BLAS Routines and Pair-HMM. In *Proceedings of Conference for Next Generation Arithmetic 2019, Singapore, Singapore, March 13–14, 2019 (CoNGA'19),* 10 pages.
https://doi.org/10.1145/3316279.3316284

1 INTRODUCTION

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. However, due to the constantly evolving set of requirements in terms of performance and accuracy, alternative number representations are proposed that claim to be a better alternative to the IEEE 754 floating point standard. One of the recent and most promising alternatives is the posit number format, as presented by John L. Gustafson [6].

In this work, we present a framework that can be used to perform posit arithmetic in hardware without intermediate normalization of computation results, resulting in more accurate final answers. The posit adder and multiplier units presented in this work take the characteristic fields that represent a posit number as input, instead of a serialized posit word. A posit normalization unit can convert the unrounded regime and fraction fields to a traditional posit word whenever desired, while applying a rounding scheme in order to take into account bits that are truncated in the process.

We present the design, implementation and evaluation of two hardware accelerators in reconfigurable logic that utilize the posit units mentioned above. The first accelerator implements level 1 BLAS operations on variable-length posit vector pairs. The second accelerator calculates the solution to a type of Hidden Markov Model (HMM) used in pairwise alignment of DNA sequences, called pair-HMM.

Both designs are based on the Fletcher framework [11], allowing easy and efficient integration with a variety of high-level software languages supported by the Apache Arrow in-memory format. Both accelerators are implemented on top of the CAPI SNAP [14] hardware platform in order to obtain coherent host-accelerator access and avoid unnecessary copy operations from host to device memory.

The work presented in this paper and the work it depends on is available in open source. We list the projects here:

- https://github.com/open-power/snap
- https://github.com/johanpel/fletcher
- https://github.com/lvandam/posit_blas_hdl
- https://github.com/lvandam/pairhmm_posit_hdl_arrow

2 MODULAR FRAMEWORK FOR POSIT ARITHMETIC

This section introduces a modular approach to building circuits to compute with posit numbers while maintaining the full accuracy in intermediate results. Conceptually this extends prior work [2] that presented a dot-product unit leveraging a (full-precision) "quire" register introduced by [7] to an arbitrary network of operations.

CoNGA'19, March 13–14, 2019, Singapore, Singapore

ACM ISBN 978-1-4503-7139-1/19/03.
https://doi.org/10.1145/3316279.3316284

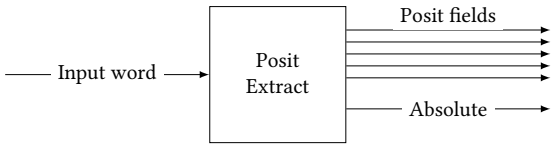


Figure 1: Schematic overview of the in- and output products for the proposed posit extraction unit.

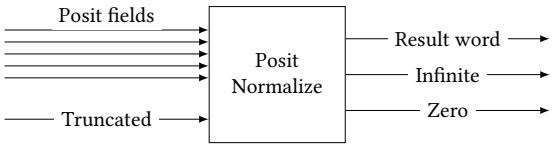


Figure 2: Schematic overview of the in- and output products for the proposed posit normalization unit.

2.1 Overall Dataflow

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, infinity/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 having the extraction, operation and normalization steps separated 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. In this section, we therefore discuss 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 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 discuss the individual steps of the data flow described above, along with the behavior of the possible posit operations that are implemented.

2.2 Posit Extraction

The posit extraction unit converts an N-bit posit value 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 is calculated based on the regime and the exponent of the input operands. As the regime accounts to a factor of $(2^{2^{es}})^{regime}$ (where *es* denotes the number of exponent bits) and the exponent value adds a factor of $2^{exponent}$, combining these factors results in the following scale factor:

$$scale = (2^{2^{es}})^{regime} \times 2^{exponent} = 2^{2^{es} \times regime + exponent} \quad (1)$$

The output of the posit extraction unit is therefore an internal structure containing the following fields:

- sign
- scale
- fraction
- inf/zero status flags

The widths of the individual fields depend on the posit configuration of the value represented (i.e. the number of exponent bits of the represented posit). Furthermore, an addition or multiplication result requires more scale and fraction bits to be stored in order to avoid truncation of any information.

2.3 Posit Normalization

In order to convert back the internal structure of posit fields (including the unrounded fraction) into a regular posit word, 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 (N − 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 taken to represent the regime and the exponent in the final posit number. The decimal accuracy can be improved by implementing a rounding scheme. 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 [5], and is chosen as default rounding scheme for the IEEE float format. Rounding is therefore performed to the nearest value. Therefore, all fraction bits that are discarded (in order to fit the final result to an N-bit value) are used to determine whether a value has to be rounded. For a tie (midway value), the value is rounded to an even number (i.e. with a zero LSB). 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.

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 ($scale \bmod 2^{es}$). 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.

2.4 Posit Adder

The posit adder is designed with a 4 or 8-stage pipeline structure, which is controlled by an input clock signal. A summary of the posit adder operations per pipeline stage is as follows:

(1) Clock in inputs, determine largest and smallest operand, calculate scale difference
(2) Match smallest operand fraction by right-shifting, add / subtract fractions
(3) Shift out hidden bit of addition result (and adjust scale)

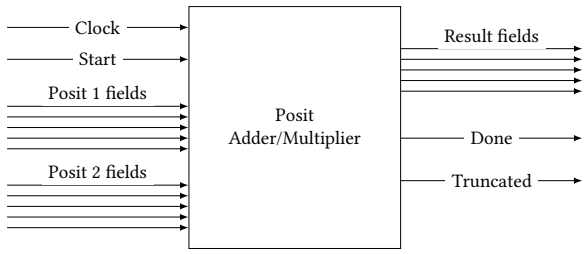


Figure 3: Schematic overview of the in- and output products for the proposed posit adder and multiplier.

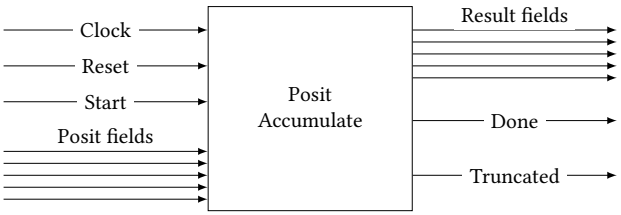


Figure 4: Schematic overview of the in- and output products for the proposed posit accumulator unit.

(4) Set output signals (result fields, inf, zero, done)

After normalization of any N-bit input operands, the extracted posit fields are passed to the posit adder, along with a start signal (which validates the output results as they are propagated through the adder).

As a preparation before performing the actual posit addition, the hidden bit is prepended to the input fraction fields. In order to match the scale (as defined in eq. (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. In this 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, which can be used by the normalization unit when performing the rounding of the truncated sum fraction (described in section 2.3) whenever a result has to be normalized.

After matching both fractions with the same scale, the fractions are added or subtracted (for unequal operand signs) using an unsigned integer adder. After detecting the location of the hidden bit, 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, among others, the unrounded scale and fraction fields, can then be used as an input operand for next operation(s) or can be normalized back into a regular N-bit posit number through the posit normalization unit described in section 2.3.

2.5 Posit Multiplier

Similar to the posit adder, the posit multiplier is implemented with 4-stage pipelining. A summary of the operations per pipeline stage for the posit multiplier are as follows:

(1) Clock in inputs
(2) Multiply operand fractions
(3) Add operand scales, shift out hidden bit of product (and adjust scale)
(4) Set output signals (result fields, inf, zero, done)

The fraction field of the input posit operands are multiplied using an unsigned integer multiplier. The scale of the output product, as defined in eq. (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. The resulting fraction product can then be put back into the normalized form `1.xxxx`. Similar to the posit adder described in section 2.4, the product of two 28-bit fractions (including hidden bit) is 56 bits wide. Therefore, the fraction bits that are discarded in the final result (by shifting in the regime and discarding leftover bits) are used to determine if the truncated fraction needs to be rounded. After truncating the product fraction bits to fit in the result posit, the exponent, regime and fraction are packed into a single N-bit posit number. This is similar to the way the final result is constructed for the posit adder described in section 2.4.

2.6 Posit Accumulator

The posit adder described in section 2.4 is designed to calculate the sum of two N-bit input posit numbers, returning an N-bit posit. Recall that the fraction of the smaller input operand is shifted in order to match the scale of both input operands. Therefore it is 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 decimal accuracy.

In order to avoid any input information to be discarded, the posit wide accumulator is proposed. The wide accumulator consists of a posit adder that is similar to the design proposed in section 2.4. The main difference is that this adder only takes one input that could, for example, be the unrounded set of sign, scale and fraction fields of a previous calculation. The second input consists of the sign, scale and fraction of the accumulated number so far. Note that the accumulated fraction is not normalized and therefore 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, which can be fed to a normalization unit (refer to section 2.3) to obtain a regular N-bit posit number.

3 POSIT BLAS ARITHMETIC ACCELERATOR

The proposed posit arithmetic accelerator is designed with modularity, usability and ease of integration 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.

To allow a multitude of software languages to make use of the accelerator, the hardware interface to the in-memory data set of vectors is generated through the Fletcher framework [11]. This

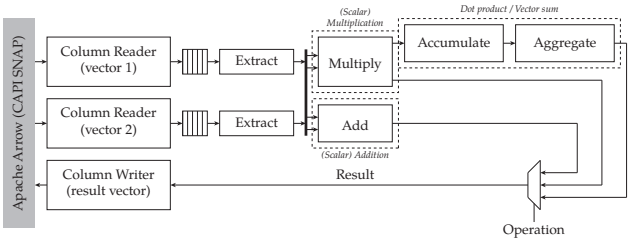


Figure 5: Schematic overview of the data flow for the posit vector arithmetic accelerator, starting from the input vector representations in Apache Arrow. Each component is annotated with the operations the component is used for.

framework allows for efficient interoperability between FPGA accelerators and software languages through the use of the Apache Arrow in-memory format [15]. Additionally, the Arrow project itself provides libraries for (at the time of writing) eleven high-level software languages to construct and consume data sets in this format. We use these two frameworks to ease the design of the hardware/software interface and prevent (de)serialization overhead when working with boxed numerical types that are common in many high-level languages (such as Python and R) and variable-length data types (such as vectors). Through this combination, it becomes easy to support high-performance Posit BLAS operations using the proposed accelerator in any of the languages supported by Apache Arrow.

A schematic overview of the accelerator structure is depicted in fig. 5, 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, Fletcher instantiates a Column Reader to read vector elements. A Column Reader enables reading from a column of an Arrow tabular data set (that in our case is filled with variable-length vectors). As the accelerator is designed to accept two input vectors, one Column Reader is instantiated per posit input vector.

On the software side, the Arrow data set containing the input vectors is initialized and passed to Fletcher that automatically makes the data set available to the Column Readers. After passing a description of the desired operation, the accelerator can now start processing the BLAS operation. The Column Readers start providing the posit vector elements to the extraction units.

Depending on the desired 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 (refer to fig. 5). 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 [16]. 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 column vector of ones.

Note that the resulting values coming from the adder, multiplier or accumulator consist of an unrounded set of posit fields. 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 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.

3.1 Implementation

For the accelerator design as discussed in section 3, 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™ XCKU060 FPGA.

Table 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.

3.2 Results

For the implementation as described in section 3.1, 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 [13]. Note that the execution time of the benchmark calculations performed in software are highly dependent on the specifications of the machine

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 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.

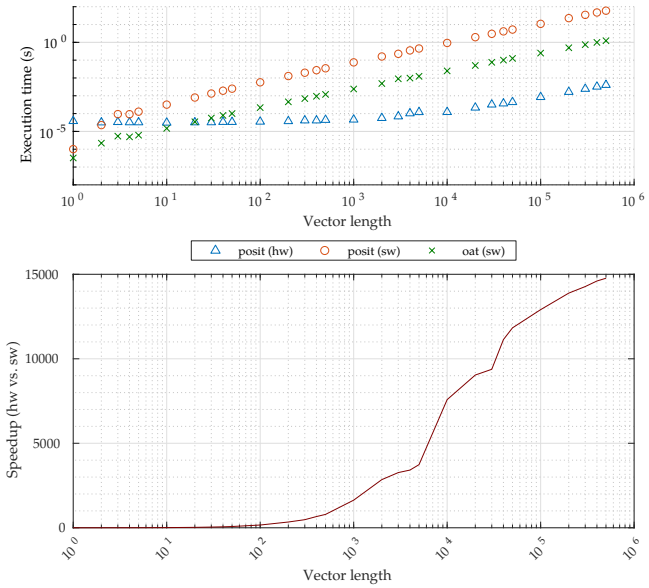


Figure 6: Execution time for a posit dot product calculation using the proposed hardware accelerator compared to dot product calculation in software for different input vector lengths. The curve illustrates the speedup of the hardware posit implementation compared to software posit emulation.

used. The machine used in this experiment is the IBM® Power Systems™ S822LC featuring two 10-core POWER8 CPUs running at 2.92 GHz.

3.2.1 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.

3.2.2 Performance. Figure 6 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× 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 7 shows the speedup compared to software calculation of the other vector operations supported by the proposed accelerator. The accelerator throughput in Mega Posit Operations Per Second (MPOPS) 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× 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×.

When comparing the proposed vector arithmetic accelerator to related work, such as the posit vector dot product accelerator presented by Chen et al. [2]. , several observations can be made. The implementation presented in that work has a working frequency of 200 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 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.

3.2.3 Decimal Accuracy. In order to compare the accuracy of calculation results for different number representations with a reference, the concept of decimal accuracy is used as proposed by John Gustafson [4]. Suppose we have the true value X and the calculated value $\tilde{X}$, which can for example be the result of a calculation using posit arithmetic. The decimal accuracy can then be defined as a measure of the number of decimals of accuracy. This measure is calculated as the log base 10 of the inverse of the *decimal error*, which is the absolute log base 10 ratio between the exact and computed value:

$$\begin{aligned} \text{decimal accuracy} &= -\log_{10}(\text{decimal error}) \\ &= -\log_{10}\left|\log_{10}\left(\frac{\tilde{X}}{X}\right)\right| \end{aligned} \tag{2}$$

As described in section 3.1, implementations of the posit vector arithmetic accelerator have been generated for the *posit(32,2)* and *posit(32,3)* configurations. fig. 8 shows the decimal accuracy of the posit dot product calculation results produced by the accelerator for both posit configurations. The decimal accuracy is compared

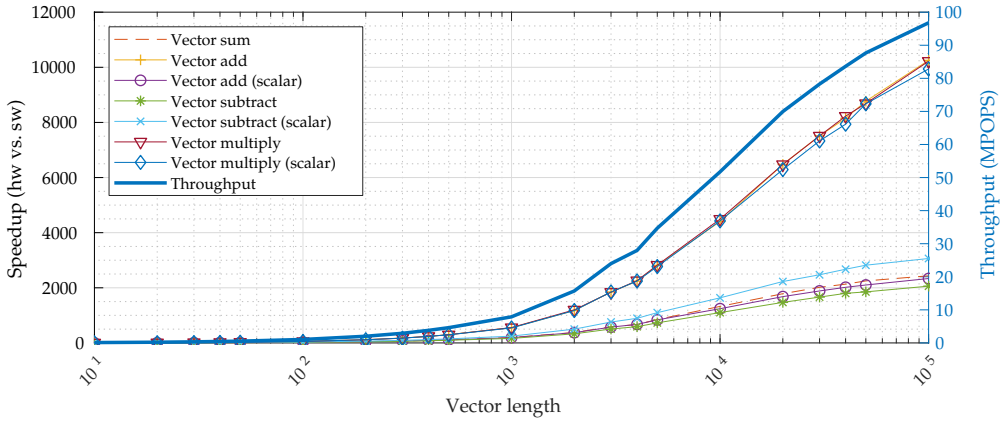
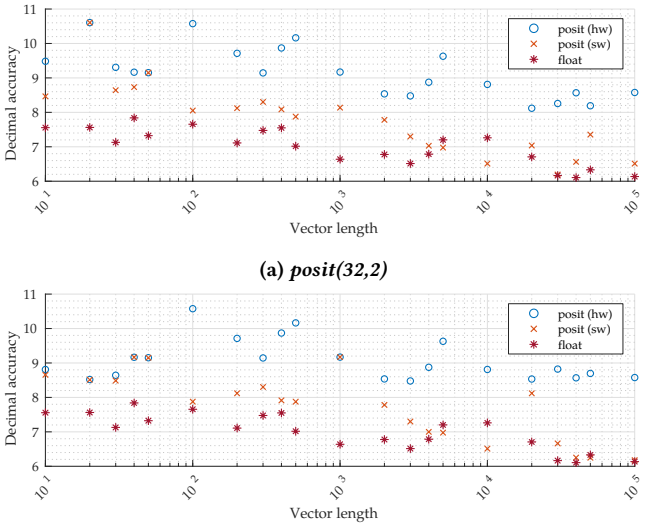


Figure 7: Speedup of the posit vector arithmetic accelerator for different vector operations and vector input lengths, compared to the execution time for posit emulation in software. The hardware throughput (in MPOPS) is depicted on the right axis.



(a) *posit(32,2)*

(b) *posit(32,3)*

Figure 8: 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.

to software calculation in the posit and IEEE 754 float format for a range of input vector lengths. In order to obtain a fair benchmark between the *float*, *posit(32,2)* and *posit(32,3)* number formats, the input vectors for this test case consist of randomly generated values. These values 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. 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 that the randomly generated number

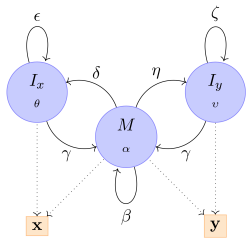


Figure 9: A pair-HMM with 3 states.

can be exactly represented in every candidate number format, accuracy loss caused by the representation of initially random numbers is omitted.

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.

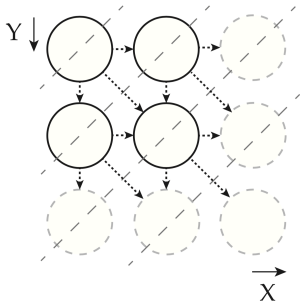
4 PAIR-HMM POSIT ACCELERATOR

A pair Hidden Markov Model (pair-HMM) is a model that can be used for the generation of probability distributions for sequences of pairs of observations. This type of HMM is particularly useful for finding the most probable alignment(s) between sequences, for example in DNA analysis when matching multiple candidate DNA *reads* with a specific *haplotype* sequence [3].

To match two strings using this method we assume that strings can be matched by either extending both strings with the same base, or extending only one or the other (corresponding to a deletion in the other string). Figure 9 shows an example Pair-HMM with three states where state Ix can emit a symbol into string X and Iy

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 2: Example sequence observations x and y and their underlying sequence of hidden states z for the pair-HMM model.



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

Figure 10: Recurrence relation and dependencies among the matrix-based calculations of the pair-HMM forward algorithm (arrows), and the progression of the calculations each cycle in a systolic array (diagonal lines).

can submit a symbol into string Y and where state M can emit the same symbol into both strings. Table 2 shows a possible sequence of states required to achieve an alignment between the two strings shown.

The computation is most easily visualized as a systolic array, with the two strings to be compared, say string X of length M and string Y of length N, along the two axes. Computing the overall probability of matching two strings with the Markov model described leads to the recurrence relation shown in fig. 10 [3] with initial conditions $M(i,-1) = M(-1,j) = I_x(i,-1) = I_x(-1,j) = I_y(i,-1) = I_y(-1,j)$, $M(0,0) = 1$, $I_x(0,0) = I_y(0,0) = 0$. Transition and emission probabilities are as indicated in fig. 9.

4.1 Accelerator Design

Our design follows the architecture of the streaming-based pair-HMM accelerator described in [12] that is based on a widely-used software implementation in [9], but with data being read from and written to Arrow tabular data sets, again leveraging interfaces generated through Fletcher. However, we implement the arithmetic of the design using the presented posit units rather than IEEE floating-point arithmetic. By using posit arithmetic and by avoiding intermediate rounding, the precision of the final results is improved.

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.

	Haplotypes haplo (8-bit)
0	base pair
	…
	base pair
1	base pair
	…
	base pair
⋮	…

	Reads read (8-bit)	probabilities (256-bit)
0	base pair	α_{diff} α_{simi} β γ δ ϵ η ζ
	…	…
	base pair	α_{diff} α_{simi} β γ δ ϵ η ζ
1	base pair	α_{diff} α_{simi} β γ δ ϵ η ζ
	…	…
	base pair	α_{diff} α_{simi} β γ δ ϵ η ζ
⋮	…	…

Table 3: Schematic overview of the Arrow schema for the Arrow pair-HMM Accelerator implementation, consisting of the columns used to feed the pair-HMM accelerator.

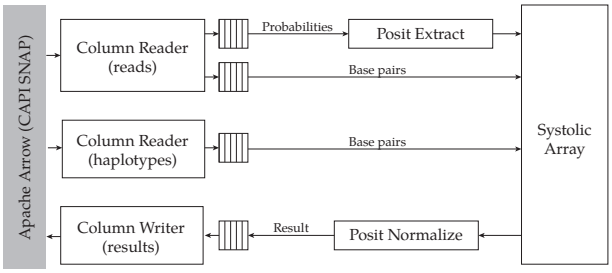


Figure 11: Schematic overview of the high-level components inside the pair-HMM accelerator core design, interfacing with Apache Arrow.

The Arrow data set designed for this implementation is depicted in table 3. As can be seen, the data set 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 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. 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.

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. 11. For the input basepair reads, a Column Reader is used in order to read the base pairs and emission/transmission probabilities from the data set. A second Column Reader is instantiated to read the basepairs from the haplotype data set. The data output of the Column Readers are fed into FIFOs. The FIFOs are controlled by a scheduler that makes sure the input data is fed into the systolic array in the correct cycle. The posit fields of the input probabilities, represented as 32-bit posit

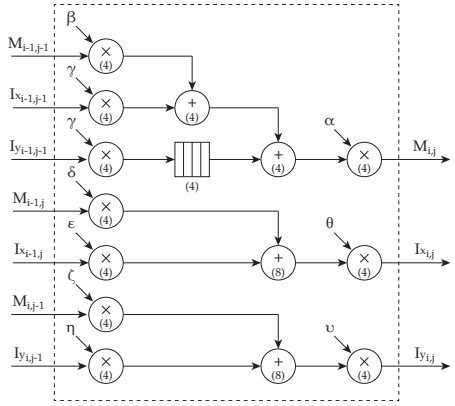


Figure 12: Schematic overview of a Processing Element. The latency (in clock cycles) of each unit is indicated between parentheses.

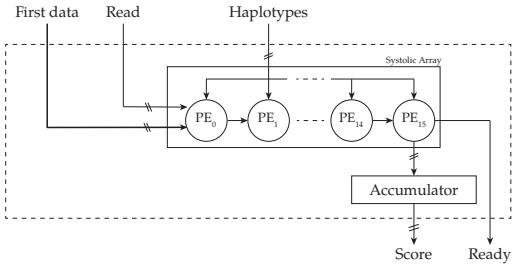


Figure 13: Overall overview of the pair-HMM accelerator core design.

numbers, are extracted using the posit extraction unit as discussed in section 2.2.

The outgoing calculation results from the systolic array, being raw posit values with unrounded fraction fields (as described in section 2.3), are then normalized. The normalized 32-bit posit words are fed into a Column Writer in order to write the results into an Arrow data set in host memory.

4.2 Accelerator Microarchitecture

The architecture proposed for this implementation is based on a fixed-size systolic array design that is optimized for maximum pipeline utilization [12]. fig. 13 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 PairHMM 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 is provided.

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)

A schematic overview of the various arithmetic operations performed per PE is shown in fig. 12. As each PE in the systolic array

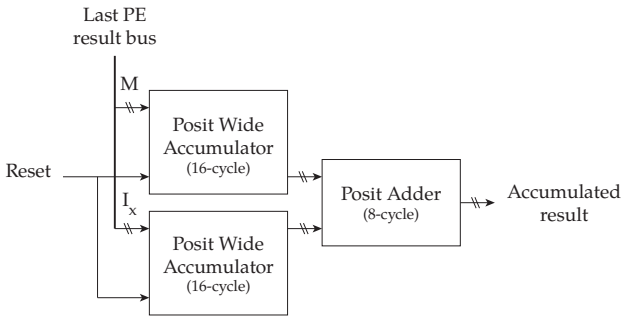


Figure 14: Schematic overview of the pair-HMM accumulation stage using posit wide accumulator units.

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.

The PE then calculates the matrix elements for the current (i, j)-position. Together with the previously calculated matrix elements, the corresponding emission and transmission probabilities are received from the previous PE.

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.

For this design, intermediate results of calculations performed inside a Processing Element (PE) as depicted in fig. 12 are kept unrounded whenever possible. The purpose 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.

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.

In order to maintain as much accuracy as possible, our design uses wide accumulators. For each matrix of the pair-HMM forward algorithm 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. 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 using wide accumulators is that more information is kept while accumulating the matrix elements of the forward algorithm. Implementing this design in the pair-HMM accelerator will result in a longer critical path in the internal circuit. Since more logic is needed in order to process the wider fractions of accumulated values, this affects either the clock frequency or latency of the design. Therefore, the decision whether to integrate the wide accumulator design into an overall accelerator design depends on a trade-off between performance and precision.

Config		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	

Table 4: 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.

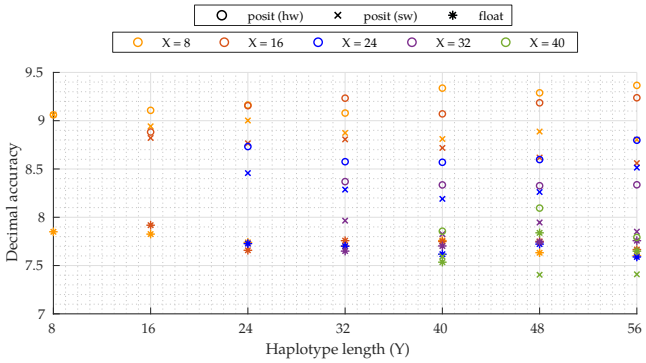


Figure 15: Decimal accuracy of the proposed pair-HMM hardware accelerator results, compared to traditional *float* computation for *posit(32,2)* . X and Y denote the read and haplotype input sequence lengths respectively.

4.3 Evaluation

An implementation of a single pair-HMM accelerator core has been generated and tested for the *posit(32,2)* configuration. We analyze FPGA resource used, decimal accuracy of calculation results and throughput performance as well as speedup compared to software implementations of the pair-HMM algorithm. The machine used in these experiments 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.

Table 4 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 Fletcher-generated logic and the Power Service Layer (PSL), required for interfacing with the host using CAPI.

Figure 15 shows the decimal accuracy of the calculation results produced based on simulation of the proposed hardware pair-HMM accelerator. The decimal accuracy of the *posit(32,2)* hardware implementations is evaluated, together with a software evaluation of the pair-HMM forward algorithm using the *float* format.

The reference calculation for determining the decimal accuracy is performed in a 100-decimal accuracy number format using the Boost Multi- precision C++ library, providing a number type with a

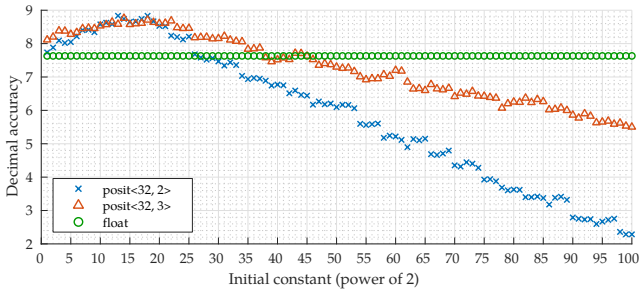


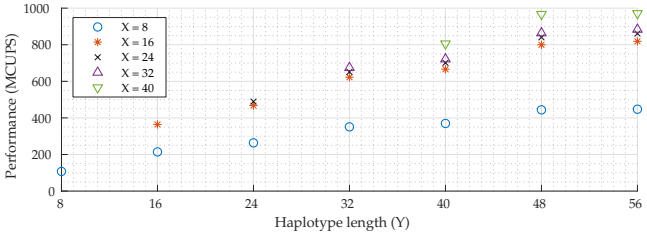
Figure 16: Decimal accuracy as a function of the initial scaling constant.

customizable number of decimal digits of precision at compile-time [1].

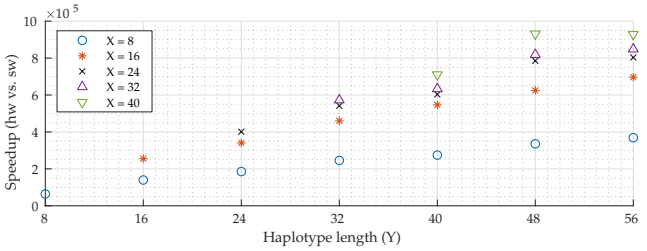
For the presented evaluations, different combinations of input sequence lengths X and Y have been tested. The initial scaling constant is set at 2^{10}. For these conditions, both the software and accelerator calculation results are performing better than the traditional *float* format for nearly every test case, with an increase in decimal accuracy ranging between approximately 0.5 and 2 decimals of accuracy.

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. 16 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.

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. 17a. 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 that 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. 17b 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.



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



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

Figure 17: 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.

5 CONCLUSION & FUTURE WORK

This paper proposed a method for creating hardware-based designs based on posit numbers that can maintain full accuracy in intermediate results, generalizing previous approaches that maintained higher-precision intermediate results in (fused) multiply-add [10] or dot-product [7] [8] calculations. A vector arithmetic accelerator for Level 1 BLAS functions of posit vectors and a pair-HMM accelerator were implemented using this framework. The accelerators are enabled with a coherent hardware-software interface. Input data is fed from column buffers represented by the Apache Arrow in-memory format, and is able to be fetched directly using the CAPI SNAP framework by the accelerator by using the Fletcher interface generator for Apache Arrow.

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× is achieved for the machine configuration as described in section 3.2. 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.

The proposed accelerator implementation 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.

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.

A second application of the proposed posit hardware framework is a pair-HMM accelerator, also operating on Apache Arrow in-memory tables and leveraging the Fletcher and SNAP frameworks to create the interfaces. With the appropriate choice of the initial constant, both the software and accelerator calculation results are 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. 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 as compared to a software implementation leveraging the posit format.

REFERENCES

[1] Boost. 2013. cpp_dec_float - 1.63.0. http://www.boost.org/doc/libs/1_63_0/libs/multiprecision/doc/html/boost_multiprecision/tut/floats/cpp_dec_float.html. (2013). [Online; accessed 2018-11-20].

[2] Jianyu Chen, Zaid Al-Ars, and H. Peter Hofstee. 2018. A Matrix-multiply Unit for Posits in Reconfigurable Logic Leveraging (Open)CAPI. In *Proceedings of the Conference for Next Generation Arithmetic*. ACM, Singapore, 1:1–1:5. https://doi.org/10.1145/3190339.3190340

[3] Richard Durbin, Sean Eddy, Anders Krogh, and Graeme Mitchison. 1998. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids* (1 ed.). Cambridge University Press.

[4] John L Gustafson. 2015. *The End of Error: Unum Computing*. CRC Press.

[5] John L. Gustafson. 2017. *Posit Arithmetic*. https://posithub.org/docs/Posits4.pdf

[6] John L. Gustafson and Isaac T. Yonemoto. 2017. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations* 4, 2 (April 2017), 71–86. https://doi.org/10.14529/jsfi170206

[7] Reinhard Kirchner and Ulrich Kulisch. 1988. Arithmetic for vector processors. In *Reliability in Computing*. Elsevier, 3–41.

[8] Jack Koenig, David Biancolin, Jonathan Bachrach, and Krste Asanovic. 2017. A Hardware Accelerator for Computing an Exact Dot Product. In *Computer Arithmetic (ARITH), 2017 IEEE 24th Symposium on*. IEEE, 114–121.

[9] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, et al. 2010. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome research* (2010).

[10] R. K. Montoye, E. Hokenek, and S. L. Runyon. 1990. Design of the IBM RISC System/6000 Floating-point Execution Unit. *IBM J. Res. Dev.* 34, 1 (Jan. 1990), 59–70. https://doi.org/10.1147/rd.341.0059

[11] Johan Peltenburg. 2018. fletcher: A framework to integrate FPGA accelerators with Apache Arrow. (2018). https://github.com/johanpel/fletcher

[12] J. Peltenburg, S. Ren, and Z. Al-Ars. 2016. Maximizing systolic array efficiency to accelerate the PairHMM Forward Algorithm. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 758–762. https://doi.org/10.1109/BIBM.2016.7822616

[13] Stillwater Supercomputing, Inc. 2017. universal: Universal Number Arithmetic. (2017). https://github.com/stillwater-sc/universal

[14] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. 2015. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development* 59, 1 (2015), 7–1.

[15] The Apache Software Foundation. 2016. Apache Arrow. (2016). https://arrow.apache.org

[16] Laurens van Dam. 2018. *Enabling High Performance Posit Arithmetic Applications Using Hardware Acceleration*. Master's thesis. ISBN 978-94-6186-957-9.