Adaptive posit tensor processing for error-free linear algebra

Theodore Omtzigt
theo@posit-research.com

***Abstract*—Posits are a tapered precision real number system that provide efficient implementations of fused dot products, enabling tensor processing with fine-grain rounding error control to deliver error-free linear algebra. We present an adaptive posit tensor processing architecture that enables hardware experimentation with posit-enabled algebras and algorithms. We will introduce posits and fused operators and summarize their benefits. The implementation in a general-purpose CPU is discussed highlighting the difficulties that fused operators create for context switch state and caches. A hardware accelerated approach solves these problem by explicitly scheduling the execution with the fused operators as constraints.**

***Index Terms*—posit, error free linear algebra, fused dot product, tensor processing, TPU, unum computing**

I. INTRODUCTION

High-performance computing techniques are rapidly changing due to sequential processing having reached a performance plateau. Hardware accelerators and domain specific processors offer better performance per Watt as compared to general purpose computational structures, but their adoption is hindered by diminished economies of scale. Only in those verticals that are constrained by power, cost, or size, is leveraging domain specific hardware accelerators an economic possibility. Fortunately, large economically valuable verticals now exist where power, cost, and size are key differentiators for computational solutions. Examples are video encode-decode in mobile devices, sensor fusion in autonomous vehicles, robotic, and embedded industrial systems, and smart sensors and analytic gateways used in the Internet of Things, and the Industrial Internet of Things. Both Google [6] and Microsoft have designed and are deploying at scale domain specific processors to aid in tensor processing specifically for Deep Learning applications in their clouds, and mobile chip makers such as Apple, Samsung, and Qualcomm all have announced inference engines. It is very clear that the leaders in the industry have progressed to building custom hardware solutions to strategically differentiate their services.

The proliferation of high-performance computing into real-time and embedded use cases has amplified a major short coming of the standard floating-point number system: floating point addition and multiplication are not associative. High-performance task-level parallel systems introduce different execution orders of the original equations causing non-deterministic reordering of intermediate results. When such systems are inspected the non-deterministic reordering makes reproduction of the failure difficult if not impossible.

Several solutions to this problem have been proposed starting in 1986 by the work of Ulrich Kulisch [5]. The basic idea in that approach is to leverage a super-accumulator that accumulates intermediate results of a computational path at full precision. The actual rounding decision is made explicit by language constructs under control of the programmer. The fundamental problem in that approach is caused by the structure of floating point: a fixed-point representation of the result of a floating point multiply requires $1 + 2 \times (2^{e_{bits}} + m_{bits})$, where e_{bits} and m_{bits} are the number of bits in the exponent and mantissa respectively. To be able to accumulate 2^k products we would add k bits to the accumulator. For single precision and double precision floats the approximate size of these super-accumulators would be 640bits, and 4288bits respectively. Modern implementations of the Kulisch idea can be found in [12].

Another approach is to use arbitrary precision arithmetic. An example is the GNU Multiple Precision Floating-Point Reliably (MPFR [10]). The upside is that very difficult computation problems in computational geometry and optimization become feasible, but the downside is that the common case is slowed down by 3 orders of magnitude.

ExBLAS [8] is a software approach that isn't as slow as arbitrary precision, but is still at least an order of magnitude slower than native execution. ExBLAS uses the super-accumulator approach coupled with a clever trick to compute the rounding error of each operation. By keeping track of how error accumulates in the basic linear algebra subroutines they are able to create reproducible results.

RepoBLAS [11] instead focuses on performance and relaxes the exactness constraint to deliver reproducible results in task-parallel execution environments.

Floating-point based arithmetic error control is complicated by the structure of the number systems. Super-accumulators grow very large due to the disproportional dynamic range of floats compared to their precision. Gustafson [1] has been working on tapered number systems to regain control over efficient and productive error control. He coined the term universal numbers, or unums for short. Unums come in several types, the Type III unums are called posits [3] and are the basis of our adaptive tensor processing architecture.

II. POSIT NUMBER SYSTEM

From [2], we learn the definition of universal numbers, or unums, for short: "Unums are for expressing real numbers and ranges of real numbers." There are two modes of operation, selectable by the user: *posit mode* and *valid mode*.

In *posit mode*, a unum behaves much like a floating-point number of fixed size, rounding to the nearest expressible value if the result of a calculation is not expressible exactly; however,

the posit representation offers more accuracy and a larger dynamic range than floats with the same number of bits. We can refer to these simply as *posits* for short, just as IEEE 754 Standard floating-point numbers are referred to as *floats*.

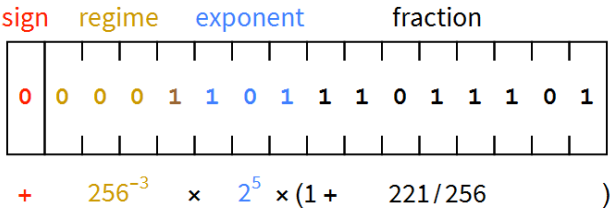
In *valid mode*, a unum represents a range of real numbers and can be used to rigorously bound answers much like interval arithmetic does, but with several improvements over traditional interval arithmetic. In this paper, we will focus on posit arithmetic exclusively.

A posit is made up of four components: sign, regime, exponent, and fraction. A posit is specified by its size in bits, *nbits*, and the maximum number of exponents bits, *es*.

Suppose we view the bit string for a posit as a 2's complement signed integer, ranging from -2^{n-1} to 2^{n-1}. Let k be the integer presented by the regime bits, and e the unsigned integer represented by the exponent bits, if any. If the set of fraction bits is $\{f_1, f_2, \cdots, f_{fs}\}$ possibly the empty set, let f be the value represented by $\mathbf{1}.f_1 f_2 \cdots f_{fs}$. Then the value of a posit is defined by the following equation:

$$x = \begin{cases} 0, p = 0, \\ \pm\infty, p = -2^{n-1} \\ sign(p) \times useed^k \times 2^e \times f, all\ other\ p. \end{cases}$$

The following figure shows these fields for a 16-bit posit with 3 exponent bits, referred to as posit<16,3>.



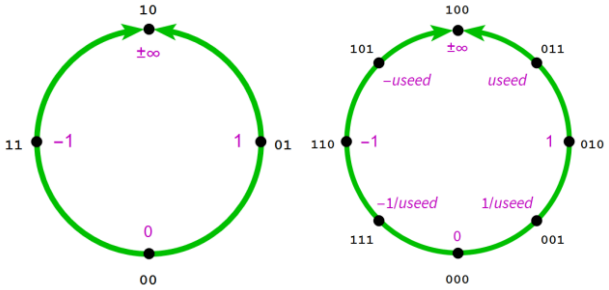
The sign bit 0, shown in red, implies that the value is positive. The regime bits have a run of three 0s terminated by the opposite bit 1, which implies the power of *useed* is -3. *Useed* is defined as $2^{2^{es}}$ and represents the scaling factor of the regime. In this example, the scale factor contributed by the regime is 256^{-3}. The exponent bits 101, shown in blue, represent 5 as an unsigned binary integer, and contribute a scale factor of 2^5. Finally, the fraction bit 11011101, shown in black, represent 221 as an unsigned binary integer, yielding a fraction value of 1.0 + 221/256. The value of this posit bit pattern is $477 \times 2^{-27} \sim 3.55393 \times 10^{-6}$.

The size of the regime and exponent fields is variable creating a tapered precision real number system, with a dynamic range perfectly symmetric around 1. The minimum and maximum positive number for a posit configuration are called *minpos* and *maxpos*. Their values are a function of the scaling factor of the regime and the size of the posit:

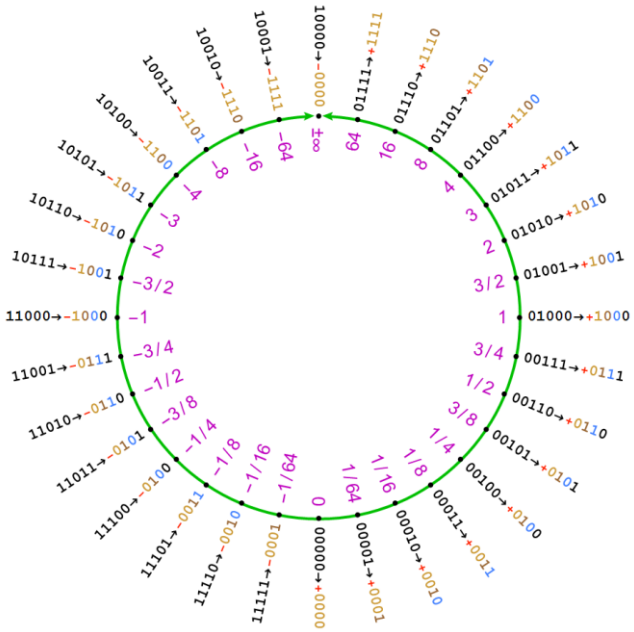
$$\{minpos, maxpos\} = \{useed^{-nbits+2}, useed^{nbits-2}\}$$

The ratio of *maxpos* to *minpos* is $useed^{2nbits-4}$, which defines the dynamic range of the posit. The posit format uses regime bits to raise *useed* to the power of any integer from -*nbits*+1 to *nbits*-1, otherwise stated, the dynamic range of a posit is an exponential of an exponential of an exponential. This allows posits to create a larger dynamic range from fewer exponent bits than IEEE floats, leaving more fraction bits available to improve the precision of a value representation.

A wonderful way to visualize the structure of a posit configuration is to realize that they derive from Type II unums that mapped binary integers to the projective reals. Projective reals wrap the real number line onto a circle so that negative and positive infinity meet at the top.



The diagram on the left represents a 2 bit posit. We move to three bits by inserting a value between 1 and ±∞. It could be any real number greater than 1; it could be *2*, *10*, *π*, or *googol*. The choice of this number *seeds* how the rest of the ring of unums is populated, to signify its importance this number was given the symbolic name *useed*. As we have seen above, for posits this value is set to $2^{2^{es}}$. Further bit expansion follows the rules that negation reflects about the vertical axis, and reciprocation reflects about the horizontal axis. The next figure shows a ring plot of values for a posit<5,1>:



III. ERROR-FREE LINEAR ALGEBRA

Posits offer higher precision than floats at the same size due to tapering. This allocates fraction bits where a typical computation needs them: around 1.0. However, the improved accuracy of posits does not provide error-free linear algebra in the same way that simply going to the next bigger float doesn't resolve numerical issues in an algorithm. The addition of a *quire* to the posit standard enables rounding control for arbitrary

computational paths and graphs. The quire is equivalent to the super-accumulator of Kulisch.

In 2008, the IEEE 754 standard [4] added a *fused multiply-add*. Fusing is defined as deferring the rounding of a computational path until the last assignment operation. For example, a floating-point fused multiply-add, or FMAC, takes the operand for addition and uses it as input to the multiplication avoiding a rounding step. This leads to less rounding error than the discrete multiply followed by the addition. However, it increases irreproducibility due to variability of compilers and hardware. Some processors do not have a fused multiply-add instruction, some processors use higher internal precision during computation, and different compilers may emit different instruction sequences. To gain control over rounding error, the rounding decision must be programmer visible.

In the posit standard the following instructions can interact with the quire:

Fused multiply-add	$(a \times b) + c$
Fused add-multiply	$(a + b) \times c$
Fused multiply-multiply-subtract	$(a \times b) + (c \times d)$
Fused sum	$\sum a_i$
Fused dot-product	$\sum a_i b_i$

The usage model for the quire is to be the intermediate accumulator for a programmer defined computational path. In code form:

```
posit<16,1> a,b,c, x;
quire<16,1> q;
q = a * b;
        … rest of the computational path
q += c;   // last step in the computation
x = q;   // final rounding step
```

This is a generalization of the single fused instructions of the IEEE standard, and this generalization is particularly valuable for the construction of math libraries. The fused dot-product is the key innovation in our adaptive tensor processor by elevating to an atomic instruction with its own stream control.

IV. POSIT QUIRES AND QUIRE SIZE

The size of a posit quire is a function of the number of bits in the posit and the size of the exponent field. In mathematical terms, the smallest magnitude nonzero value that can arise after a multiply is $minpos^2$. Every other product is an integer multiple. The largest possible product value is $maxpos^2$, and thus a fixed-point integer big enough to hold these extremes is:

$$\frac{maxpos^2}{minpos^2} = useed^{2\times2\times(nbits-2)} = 2^{(4nbits-8)2^{es}}$$

As with the discussion of the super-accumulator in the introduction, we need to accumulate some non-trivial number of these products, and for k additional bits you would be able to accumulate 2^k products. A value of $k = 30$ would accommodate roughly a billion products. For custom hardware we can dial this number in to perfection. Add a final sign bit and the size of a posit quire is given by the equation: $(4nbits - 8)2^{es} + 1 + 30$.

The following table shows the quire sizes for different standard posit sizes and we juxtapose the size of an equivalent floating-point super-accumulator for comparison.

POSIT			FLOAT		
nbits	es	quire	nbits	ebits	quire
8	0	64	8	2	21
16	1	256	16	5	87
32	2	512	32	8	647
64	3	2048	64	11	4295
128	4	8095	128	15	65763

Table 1. Quire sizes for posits and floats

As a rule of thumb, a posit has roughly the same precision as a float twice its size, so 32-bit posits compete with 64-bit floats, and the posit quire is a factor of 8 smaller.

V.GENERAL PURPOSE QUIRES

To enable quires for explicit rounding control they need to be visible to the programmer. For traditional load-store instruction set architectures (ISA) this implies that we need to create a special quire namespace and connect that namespace to the individual instructions that need to participate.

Given the size of the quire, loading and unloading it to memory is a long-latency operation that would kill any performance of a traditional arithmetic pipeline. A fused multiply-accumulate pipeline typically runs at 1 clock cycle throughput, but a memory load of a 512bit value is measured in hundreds of clocks. To sustain these throughputs, all high-performance linear algebra algorithms use some form of blocking to prime the caches with the next set of operands. Depending on the DRAM parameters and the cache line size, a typical optimized data flow would bring in 2D blocks of 8-16 lines. If we design a quire register set and ISA for fused dot products, to support the blocking of memory, we would need to create a register file of at least 8 and preferably more addressable quire registers. For 64-bit floats, an 8-register quire extension would require roughly 32k, or the size of a typical L1 data cache. But even more problematic than its size is that this quire register file would be part of the context switch state of a general-purpose processor. An additional 32k of context switch state is insurmountable for a CPU. It is thus more likely that hardware accelerators and domain specific processors are the only computational structures that will be able to offer error-free linear algebra.

VI. HARDWARE ACCELERATION

Google's Tensor Processing Unit™ [6] is a notable example of the benefits of custom hardware acceleration for tensor operators. They focused on the dense matrix/vector operations present in neural network training and inference workloads. The data flow that this class of tensor operators need is centered about the weight matrices. In our case, we want to operationalize the fused dot-product to deliver error-free linear algebra, and as we have seen in the previous section, this

requires special handling of the quire in conjunction with blocked data movement to and from memory.

Figure 1 shows the error-free posit-based linear algebra hardware accelerator architecture.

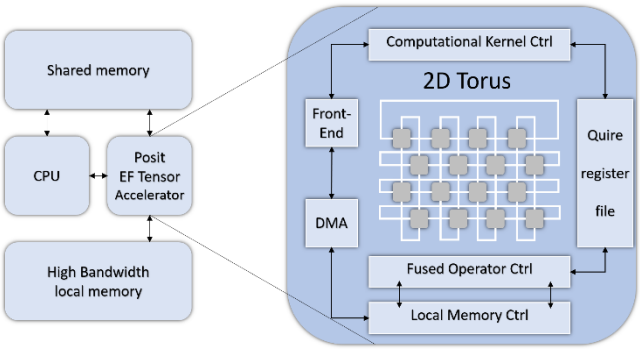


Fig. 1: Error-free Tensor Accelerator

The error-free tensor processor is built around a scalable torus fabric of posit arithmetic units. The Front-end and DMA engines orchestrate tensor data flow movement to support fused dot-product streams. They use a block-oriented memory access pattern to improve DRAM efficiency and access latency. As blocks arrive at the fused-operator controller, the contained fused operator operand sets identify the respective quires that need to receive the intermediate results. Under the direction of a computational kernel, the DMA, fused operator controller, and quire register file coordinate a specific stream schedule that leaves the quires stationary until they are assigned to their final destination at which point they are evicted from the quire register file. This avoids having to move quires to and from memory: instead, we move dot-product streams past the quires.

VII. Error-free Distributed Memory

The error-free posit tensor accelerator was designed to scale up and down to accommodate large-scale HPC applications and low-latency real-time embedded applications. This is accomplished by a traditional distributed memory design with an additional fine-grain data flow accelerator network. The scalable cluster design is show in Figure 2. We leverage a fast serial processor to address workloads that have a significant serial component and are governed by Amdahl's Law. The design couples that with a scalable tensor hardware accelerator to address workloads that are governed by weak scaling constraints. The CPU side is connected via an IP network to support coarse grain distributed memory operations to support very large in-memory data structures and global transformations such as global addressing mods and reductions. On the tensor accelerator side, computation is organized in fine-grain data flow to support the quire accumulation The accelerators are also connected by a network, but instead of an IP network, this network is a mesh network to support fine-grain data flow computations and transformations such as sorts and transposes.

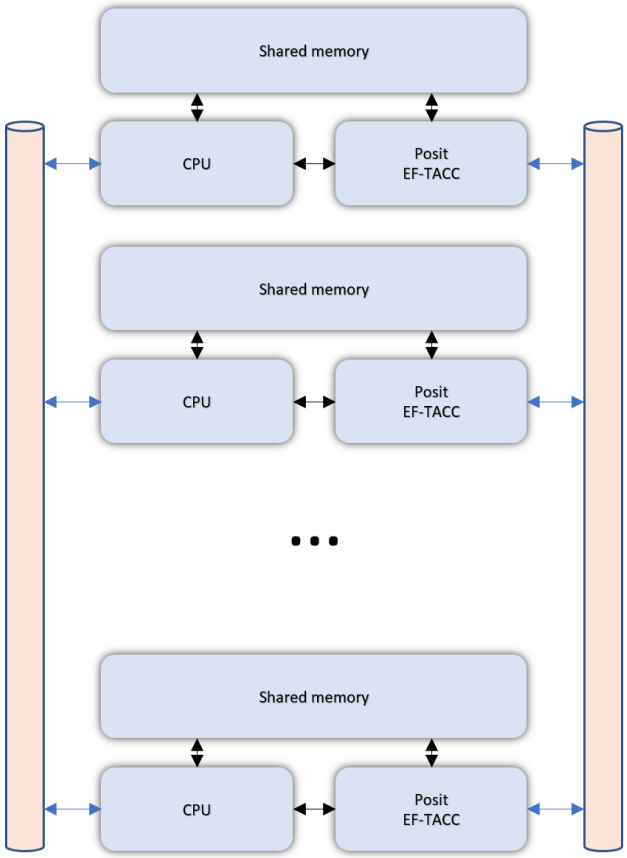


Fig. 2: Distributed memory error-free linear algebra cluster

VIII. Conclusion

We have presented a scalable, error-free linear algebra hardware acceleration architecture based on posits. The fundamental constraint in delivering error-free computation is an explicit management of the accumulation of intermediate results in programmer visible quires. These quires are reasonable for posit arithmetic, but grow very large for floating-point arithmetic. This creates an insurmountable constraint for incorporating quires into a general-purpose CPU as the context switch state becomes too large to be practical. This leads to the proposal to consolidate the quire management to domain specific processors and hardware accelerators, and we presented such an architecture centered about fused dot-products.

Acknowledgment

This work is heavily influenced by the work and publications of John Gustafson, currently at National University of Singapore. Hardware design experimentation has been provided by Calligo Technologies in Bangalor, India, and the cluster design was provided by Stillwater Supercomputing, Inc., California. The incorporation of posits into MTL4, a high-performance distributed memory and GPU acceleration math library was provided by Simunova AG, in Germany.

REFERENCES

[1] John L. Gustafson, *The End of Error: Unum Computing,* Chapman and Hall/CRC Press, February 2015.

[2] John L. Gustafson, "Posit Arithmetic," Mathematica Notebook describing the posit number system, 30 September 2017.

[3] John L. Gustafson, Isaac Yonemoto, *Beating Floating Point at its Own Game: Posit Arithmetic*. April, 2017, online http::posithub.org

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

[5] Ulrich W. Kulisch and W.L. Miranker, "The arithmetic of the digital computer: a new approach", SIAM Review, Vol. 28, No. 1, March 1986.

[6] Norman P. Jouppi et. al., *In-Datacenter Performance Analysis of a Tensor Processing Unit™.* Toronta, Canada, ISCA 2017.

[7] Y. Uguen and F. de Dinechin, "Design-space exploration for the Kulisch accumulator," March 2017, working paper or preprint [Online], Available: https://hal.archives-ouvertes.fr/hal-01488916

[8] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat, "ExBLAS: Reproducible and Accurate BLAS Library," Jul. 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01202396

[9] BLAS (Basic Linear Algebra Subprograms). [Online]. Available: http://www.netlib.org/blas/

[10] L. Fousse, G. Hanrot, V. Lef`evre, P. P´elissier, and P. Zimmermann, "MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding," ACM Trans. Math. Softw., vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[11] P. Ahrens, H. D. Nguyen, and J. Demmel, "Efficient reproducible floating point summation and BLAS," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-229, Dec 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-229.html

[12] Jack Koenig, David Biancolin, Jonathan Bachrach, Krste Asanovic, "A Hardware Accelerator for Computing an Exact Dot Product," IEEE 24th Symposium on Computer Arithmetic (ARITH), 2017.