2018 IEEE International Parallel and Distributed Processing Symposium Workshops

Hardware Implementation of POSITs and Their Application in FPGAs

Artur Podobas
Tokyo Institute of Technology
Tokyo, Japan
Email: podobas.a.aa@m.titech.ac.jp

Satoshi Matsuoka
Tokyo Institute of Technology
Tokyo, Japan
Email: matsu@is.titech.ac.jp

***Abstract*—The failure of Dennard's scaling, the end of Moore's law and the recent developments regarding Deep Neural Networks (DNN) are leading computer scientists, practitioners and vendors to fundamentally revise computer architecture to look for potential performance improvements. One such candidate is to re-evaluate how floating point operations are performed, which have remained (nearly) unchanged for the past three decades. The POSIT numerical format was introduced in mid-2017 and is claimed to be more accurate, have a wider dynamical range and fewer unused states compared to the IEEE-754. However, the hardware implications of migrating to a POSIT format are unknown. In this paper, we present our results regarding POSITs and their implementation on Field-Programmable Gate-Arrays (FPGAs). We designed a tool that automatically generates and pipelines POSIT operators that can be used as drop-in replacement in processing units or in High-Level Synthesis tools (e.g. Intel FPGA SDK for OpenCL). We empirically quantified the performance and area overhead of our POSIT implementation compared to two well-known IEEE-754 implementations, and show that our units can operate at comparable frequencies. We finally show how we can integrate our POSIT hardware into existing Intel FPGA SDK for OpenCL data-paths, enabling software programmers to easily continue to evaluate POSITs.**

I. Introduction

The ending of Moore's law [1], the failure of Dennard's scaling [2] and the recent explosion in neural networks is challenging our view on computer architecture. The race for faster and larger neural network training and inference is encouraging system architects to reconsider many previously unchallenged design decisions; reconsiderations that can benefit the computing society as a whole. One such candidate is to look at how we compute with floating-point numbers.

Changing the floating-point format offers a relatively non-invasive way to increase system performance, and reducing the floating-point format-size is beneficial for several reasons: **(I)** reduction in format-size leads to a reduction in silicon real-estate cost for Floating-Point Units (FPUs), allowing the freed silicon to be spent on more compute (wider vector, more cores) or on-chip store (larger caches), **(II)** reduction in format-size leads to more compute per unit bandwidth (more values fetched per unit bandwidth), and **(III)** reduction in format-size leads to lower power- and energy-consumption of FPUs [3].

Driven by Deep-Learning (DL), recent architectures have all come to realize alternative- or mixed-protocols for floating-point computations. NVIDIA Volta-100 [4] Graphics Processing Units (GPUs) offer a tensor unit in a mixed-precision mode, where floating-point multiplication is performed using half-precision IEEE-754 arithmetic, and reduction is performed in single-precision. Intel's upcoming NERVANA Deep-Learning accelerator [5] introduces a format where the exponent is shared within tensor-operations. Microsoft's BrainWave [6], Xilinx [7] and several other actors are proposing reduced-precision fixed- or floating-point representations. These are all exciting new formats, but most are tailored towards that single application domain called Deep-Learning.

In 2017, Gustafson et al. [8] proposed a new *general-purpose* floating-point representation called *POSIT*s. POSITs are claimed to offer a wider dynamic range, be more accurate, and contain fewer unused (e.g. Not-A-Number) states than IEEE-754 formats.

Whether the proposed POSIT format will supersede or complement the current IEEE-754 depends on many aspects. One of these aspects is the impact POSITs have on modern hardware, which is currently unknown, making it hard to reason about the gains and expenses that POSITs offer compared to IEEE-754. It is here that we claim our contribution:

- we describe and design the (to our knowledge) first POSIT implementation for three commonly used arithmetic operations on FPGAs,
- we empirical quantify the performance and area comparison between POSITs and two different implementations of the IEEE-754 for addition, subtraction and multiplication,
- we show how our components can be integrated into High-Level Synthesis (HLS) tool-flow, simplifying integration and evaluation of POSITs in existing High-Performance Computing infrastructure, and
- we implemented an OpenCL prototype linear algebra library based on POSITs, and quantify the performance we achieve with it and compare it to an existing, C++ based, POSIT emulation library.

II. The **POSIT** format

The IEEE-754 standard describes how to store, format and compute with real-valued numbers. The current version supports five different widths, ranging from the small 16-bit `half`-precision format, all the way up to a 256-bit `octuple`-precision format. The width of the format largely depends on the application at hand; deep-learning training is performed

978-1-5386-5555-9/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPSW.2018.00029

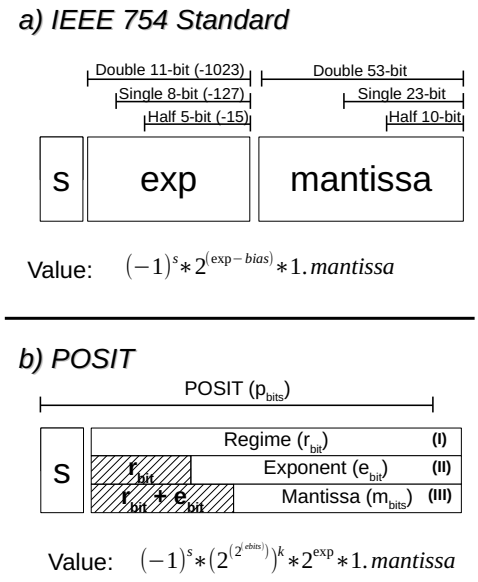

Fig. 1. Visual illustration between the IEEE-754 (a) and the POSIT (b) format. The similarities between the formats end with the *sign*-bit being at the same (MSB) place. Note how the exponent and mantissa fields change in size as a function of the regime– in some cases, there is no mantissa (**II**) or exponent (**I**)).

using single-precision while image manipulation often allows for a reduction in precision. Some applications, such as GROMACS [9], even mix different precisions for performance reasons.

In 2015, John L. Gustafson wrote "The End of Error: Unum Computing" [10] where he introduced an alternative, arguably more precise, way of representing real numbers in computer arithmetics. This initial candidate, originally called `unum` (from now: `unum`-I), introduced a variable-length format that includes a *uncertainty*-bit (called the *u* bit) that indicates whether the number is precise or within an interval. While the `unum-I` potentially offered increased numerical accuracy, it was also criticized for being hardware-*unfriendly*, mainly due to the variable length of the format. The silicon cost of `unum-I` has also recently been quantified [11], [12].

Following the criticism of `unum-I`, Gustafson et al. [13] proposed `unum-II`. The new proposal contained many appealing mathematical properties and featured an elegant (and fast) way of calculating the exact reciprocal. However, the proposed `unum-II` required the use of pre-computed look-up tables to perform arithmetic, making `unum-II` impractical for anything but small data-width sizes.

Relaxing some of the mathematical properties of `unum-II` finally led to *POSIT*s [8]. POSIT is introduced to be a hardware-friendly version of the `unum`-family of numerical formats, and a "drop-in" [8] replacement for the IEEE-754 format.

A. POSITs vs IEEE-754

Figure 2:a provides an overview over both the traditional IEEE-754 (a) and the new POSIT (b) format. The IEEE-754 format comes in multiple sizes (three are shown in the figure) but the form is similar across all formats: a sign bit indicates if the number is negative (set) or positive (unset), a constant number of exponent bits together with an implicit (format-dependent) bias indicate the range of the number, and the remaining bits are allocated to the mantissa, which when combined with an implicit *1* yields a number between 1.0 and 2.0. Note that for the IEEE-754 format the bit-fields remain constant, *independent* of the numerical value represented. A value in the IEEE-754 floating format is thus calculated as:

$$val = (-1)^{sign} * 2^{exp-bias} * 1.mantissa \quad (1)$$

Figure 2:b shows the corresponding POSIT format. The POSIT is of size p_{bit} and the format has four bit-fields: a sign bit indicating positive or negative numbers, a *regime* (r_{bit}) and *exponent* (e_{bit}) field that together represent the scale factor, and finally a mantissa (m_{bits}). The POSIT format is defined by two (preset) variables: the total p_{bit} and the maximum width of e_{bit} (sometimes called es). There are three important differences between the POSIT format and the IEEE-754:

- Out of the four fields, only the *sign* is constant– it is still a variable format, albeit with a fixed total size,
- the regime is *encoded*, and the size and content of the exponent (e_{bits}) and mantissa (m_{bits}) is unknown before r_{bit} is known,
- rather than having a single exponent term, the POSIT format uses two exponents two determine the working range of the number; the exponent is treated as an unsigned term and the regime is treated as a signed term. The exponent and the regime can be concatenated to yield a single signed value representing the full working range. Unlike IEEE-754, there are no *subnormal numbers*.

A value in the POSIT format is calculated as:

$$val = (-1)^{sign} * (2^{2^{e_{bits}}})^k * 2^{exp} * 1.mantissa \quad (2)$$

III. Related Work

The amount of IEEE-754 research that has been performed since its inception in 1980s is nearly uncountable, and has led to the optimized IEEE-754 designs that we enjoy today. Among the more influential FPGA-related IEEE-754 work has been performed by Dinechin et al. through their FloPoCo [14] framework. FloPoCo allows users to automatically generate and pipeline various IEEE-754 semi-compliant operators. FloPoCo supports far more operators than we examine in the present paper. The only downside with FloPoCo (and other practical IEEE-754 designs such as those by Xilinx and Intel) is that they ignore the sub-normal part of computations. The same group also published early results regarding the implementation of `unum-I` and found it to have a significant area-overhead compared to IEEE-754 [12]. While on the topic

on IEEE-754, it is worth mentioning that there is a recent trend to describe the floating-point operators using High-Level Synthesis (HLS) rather than low-level VHDL and Verilog. For example, Bansal et al. [15] shows that the difference between automatically generated, low-level floating-point cores and those described in C with LegUp is not as large as previously thought. Similar work was performed by DiCecco et al. [16] using Xilinx HLS tools, here with the focus on deep-learning.

Recently, Glacer et al. [11] taped out a `unum-I`-based ALU in 65nm technology, consider the first ever. They complement the work of Bocco et al. [12], and state that the complexity of the data-path is "significantly" increased, which indirectly encouraged the work of the present paper. Hou et al. [17] implemented `unum-I` support in FPGAs for a subset of operators, showing its application on a 16-point FFT. Their result indicates that `unum-I` have a high information-to-bit ratio, and conclude that it should be a candidate to replace IEEE-754.

Peer-reviewed information regarding POSITs is scarce, as the format is very new (introduced in mid-2017). However, there is on-going work that converts a C# POSIT library to FPGAs [18].

IV. POSIT Arithmetics in Hardware

To reason about the hardware aspects of the POSIT proposal, it must first be realized in hardware. Currently – to the best of our knowledge – there is no such description of POSIT hardware in literature.

Before describing our POSIT implementations, we need to provide bounds for the various fields in the POSIT format. Recall that a POSIT number of size p_{bits} has four bit-fields: the sign s_{bit}, the regime r_{bits}, the exponent e_{bits} and the mantissa m_{bits}. Out of these four bits, only the s_{bit} is constant– all other fields and their size is a function of the regime (r_{bits}), which is variable in size and is encoded.

Despite being non-constant in their size, we can provide (upper-) bounds regarding their field-size (called $f()$). These bounds help us in deciding the bit_vector sizes in our hardware later on. The bounds are:

$$f(r_{bits}) : 2 \leq r_{bits} < p_{bits} \tag{3}$$

$$f(e_{bits}) : 0 \leq e_{bits} \leq min(e_{bits}, p_{bits} - 1 - f(r_{bits})) \tag{4}$$

$$f(m_{bits}) : 0 \leq m_{bits} < p_{bits} - 1 - f(r_{bits}) - f(e_{bits}) \tag{5}$$

There are several interesting points that these bounds tell us. We note that the regime-bits can potentially occupy nearly the entire width of the POSIT; in such scenarios, both the exponent and mantissa bits are implicitly set to zero. Similarly, despite the fact that the number of exponent bits are fixed for a certain POSIT format, even these can eventually end up not being present in the representation.

The bounds on the mantissa bits are arguably the most important and impactful, since they decide the magnitude of the basic integer operations within the compute units. Field-Programmable Gate-Arrays often have hard Digital Signal Blocks (DSPs) inside the fabric to accelerate operations that normally would occupy much of the reconfigurable logic. Among the common uses for DSPs is to calculate the product of two integer numbers, where the DSP can be configured [1] to perform three small multiplications (9-bit x 9-bit), two medium multiplications (18-bit x 18-bit), or one large multiplication (27-bit x 27-bit). Hence, despite the current lack of a standard for POSITs, it is important to note that the size of the e_{bits} directly influence the size of the operands in POSIT arithmetics; fewer e_{bits} will increase the operand-width in our floating-point unit.

We continue by describing our POSIT addition, subtraction and multiplication, starting with how to *decode* and *encode* the POSIT format.

A. Decoding POSIT Numbers

Unlike IEEE-754, whose fields are easily extracted based on their sizes, POSIT values need to be decoded. Only the regime (r_{bits}) is encoded and, after it has been figured out, the remaining exponent and mantissa are easily extracted. We call the *decoded* regime k.

Our decode component is shown in Figure 2:a. The regime is encoded similarly to one-hot encoding. We identify the first bit after the sign bit and call it $b_{p_{bits}-2}$. After that, we iterate through the remaining bits for as long as they remain the same (equal to $b_{p_{bits}-2}$), and stop at the first bit that differs. We count the number of consecutive bits we found and call the count k'. If we were counting unset bits (`'0'`) then the regime is negative ($k < 0$), otherwise the regime is positive and we decrease it by one (the leading bits `'10'` implies $k = 0$):

$$k = \begin{cases} k' - 1, & \text{if } b_{p_{bits}} =' 1' \\ not(k') + 1, & \text{otherwise} \end{cases} \tag{6}$$

Practically we implement the decoder as a single count-leading zero (CLZ) module. We divide the numbers into chunks of 4-bits and construct an adder tree. A node in the adder tree will only perform the addition if: (**I**) the most-significant bit of the left-most operand is set for even accumulator sizes, or (**II**) the left-most operand is set to -1(signed) for odd size, (**III**) otherwise it only propagates the left operand. The accumulator is increased by one bit for every level in the tree. We use a single CLZ module to conserve area, requiring us to make sure we are counting zeroes by inverting the number should the MSB be set. After decoding the regime, we right-shift the bits that were consumed by the CLZ module, revealing the e_{bits} of the exponent. We extract the exponent, shifting the operand right by another $e_{bits} - 1$ (this shift is constant), forcefully setting the most-significant bit (the implicit '1') of the shifted expression and obtain the final mantissa. At this point, we are finished decoding.

B. Encoding POSIT Numbers

Encoding a POSIT number – assuming that we know its sign, regime, exponent, and mantissa – is shown in Figure 2:b.

[1] Xilinx and Intel FPGA DSPs are different. This work targets Intel FPGAs, but we are already porting our tool to Xilinx (Kintex Ultrascale+) FPGAs

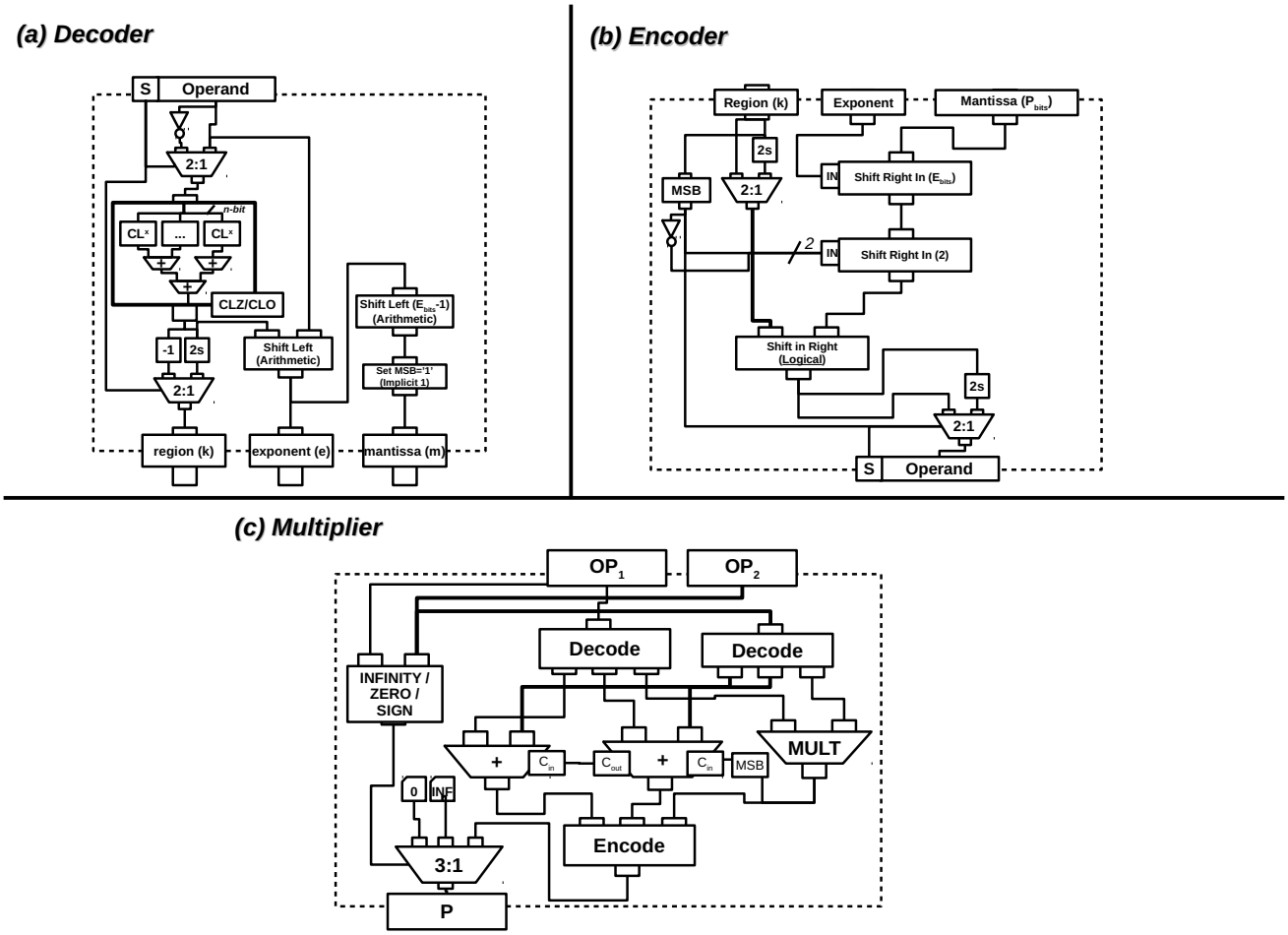

Fig. 2. Component diagram for decoding (a) , encoding (b) and multiplying (c) POSIT numbers.

We start by shifting-in (from the left) the exponent-bits into the mantissa. We continue by identifying the sign of the regime k, which we concatenate with the inversion of the sign and shift-in; if the sign is positive ('0'), then we shift in '01' and if negative ('1'), we shift in '10'. Doing this preprocessing is important as it simplified the final generation of the regime.

Finally, we find the magnitude ($|k|$) of the regime and *logically* right-shift the product $|k|$ number of steps, replicating the MSB bit to encoded the value. We finalize the encoding by two's complementing the product if the sign-bit is set, right shift in the sign bit, and finally emitting the decoded values.

C. Multiplication

Figure 2:c outlines the block diagram over our multiplication POSIT operator. It accepts two inputs called OP_1 and OP_2 in the POSIT format and it outputs the product P in POSIT format. We start by detecting corner cases, including if the result should be ∞ or zero; these single-bit signals are propagated down the entire pipeline. We also exclusively-or the sign bits (as done in normal IEEE-754 format) to determine the sign of the final value. After preprocessing the operands we decode both input operands, which is done in parallel. After both operands are decoded, the product is computed similarly to that in IEEE-754 computations. We multiply both operands to obtain the product and shift the product right one step if the MSB of the product is set (to make sure the implicit bit is 1). We also add both operands exponent and regime– any overflow from the exponent summation is carried into the regime summation. We then finish the operation by encoding the resulting sign, regime, exponent, and mantissa.

D. Addition and Subtraction

Addition and subtraction is conceptually very similar to how they are performed in the IEEE-754 format, but with the extensions of decoding and encoding the value. The adder consists of two input operands OP_1 and OP_2, one output SUM, and a single-bit input that regulates addition or subtraction. We start by identifying if the expect operation is a subtraction; if it is, we invert (by taking 2s complement) the second operand (OP_2) and proceed with adding them ($X - Y = X + (-Y)$). Similar to the multiplication, we start by preprocessing the operands. We identify possible ∞ values and if both operands

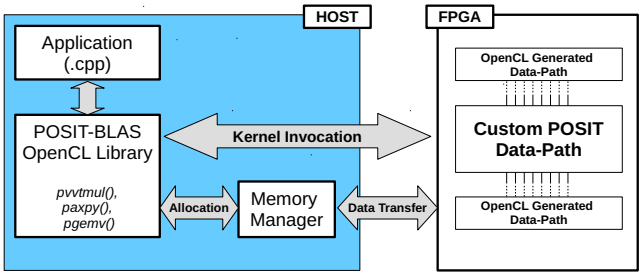

Fig. 3. Our infrastructure for testing integration of POSITs into Intel OpenCL data-paths.

are equal to each other ($OP_1 == OP_2$) to force through a sum that is either zero or ∞. After preprocessing we decode each operand and the remaining parts follows a typical IEEE-754 flow: we force the first operand to be strictly larger than the second one ($OP_1 > OP_2$), switching their values and sign to ensure this. We then compute the difference between the regime and exponent between the two operands and normalize (shift) OP_2 to that of the larger OP_1. We then continue to either add or subtract both vectors together. To finalize the operation, we locate the first set bit in the sum: if the location of the first set bit indicates an overflow, we increase the exponent by one; if the location of the first set bit is not the implicit bit we decrease the exponent and regime, otherwise we do nothing. The number of leading unset bits are used to left-shift the sum in order to reveal the implicit 1 bit. Finally, we encode the sum in the same way as in the multiplication case.

E. POSit GENerator

We implemented a prototype tool named POSit-GENerator–POSGEN. The tool itself is inspired by FloPoCo in that it automatically generates POSIT operations based on users constraints. A user supplies POSGEN with the operator he/she wants to create, the total format bit width (p_{bit}) and exponent width (e_{bit}). Our tool takes the user's specification and automatically constructs and pipelines the hardware. Being a prototype, we have focused the pipelining effort on 32-bit POSITs (albeit, any bit-width is handled by our tool)–future work will focus on optimizing the pipelining strategy for other bit-widths. The output of our tool is a synthesizable VHDL description, and an Intel OpenCL SDK for FPGA XML-specification to simplify POSIT integration into existing OpenCL data-paths.

V. Experimental Setup

All experiments were carried out with and synthesized against the Stratix V DE5-Net FPGA development kit. The board consists of a Stratix V FPGA (part: 5SGXEA7N2F45C2) interfacing two external DDR3 memories. We used Quartus 17.0.1, Intel FPGA SDK for OpenCL 17.1.1 (Build 593) and used Terasics board support package (BSPs) version 16.1 for OpenCL.

We compared our addition, subtraction and multiplication POSIT designs against those automatically generated by Intel's custom floating-point (`ALTERA_FP_FUNCTIONS`) generator [19], as well as against the FloPoCo generator [14]. For both FLoPoCo and the Intel IP generator we set the target frequency as 500 MHz (27-bit x 27-bit DSP maximum frequency for Stratix V). For both Intel's generator and FloPoCo we synthesize half- and single-precision, while for our own design we synthesize from 10-bit up-to 32-bit in steps of two bits for three different sizes of e_{bits}. The units were synthesized in isolation with virtual pins, and we report the *unrestricted* f_{max} in MHz [2].

For experiments involving integration of POSITs into Intel OpenCL and for the linear-algebra functions (`pvvtmul`, `paxpy`, `pgemv`), we used the infrastructure shown in Figure 3. The infrastructure we created consists of our linear-algebra library (**PBLAS**) that connects to the FPGA through the OpenCL API. Our library has a memory manager that keeps track of memory regions currently existing on the FPGAs and their state (dirty, clean) to minimize transfer overhead.

VI. Results

A. Area, Frequency and DSPs comparison

Figure 4 shows the characteristics of our POSIT cores compared to those of FloPoCo's and Intel's floating-point generators. We compared them with respect to: unrestricted f_{max}, number of pipeline stages, occupied ALMs and DSPs used.

Figure 4:a shows the comparison for the multiplication. We see that the frequency of our generated cores matches those generated by Intel and FloPoCo, albeit the IEEE-754 cores operate slightly higher. Independent of the exponent size(e_{bits}), our cores consistently reach between 500 and 604 MHz throughout a majority of the bit-width spectrum. The main difference lies in the number of pipeline stages; our cores have been less tuned (and too aggressively) pipelined, which yields more than twice as many pipeline stages compared to both the IEEE-754 cores; also, note the elegance of FloPoCo that, compared to that of Intel, has significantly fewer pipeline stages. The number of pipeline stages for our design remains between 33 and 38, primarily since we have been focusing on the 32-bit version of POSITs. Figure 4:b shows the ALM usage, and we see the impact of the decoder and encoder here; the POSIT cores are current significantly more area-expensive compared to both Intel and FloPoCo. Note, however, that the amount of DSP blocks that the POSIT core occupy is lower than that of FloPoCo (that needs two DSPs blocks); for DSP-bound designs this means that we can pack twice as many POSIT multipliers over using FLoPoCo. There is one exception: the 32-bit POSIT multiplier with an exponent size (e_{bit}) size of less than three will increase the mantissa to the

[2]To meet final timing constraints, we are indeed limited by maximal frequencies of DSPs, BRAMs etc. However, this allows us to reason about how much extra slack users or tools have to add logic or re-routing paths.

TABLE I
SUMMARY OVER LINEAR ALGEBRA ROUTINES IMPLEMENTED (α,β = CONSTANTS

Name	Function	Group
`pvvtmul`	$A = xy^T$	vector-vector mult.
`paxpy`	$y = \alpha x + y$	vector-vector add
`pgemv`	$y = \alpha Ax + \beta y$	matrix-vector mult.

point where a whole extra DSP block is needed; this is in line with the bounds we formulated in section IV before.

Figure 4:c shows the comparison of the addition/subtraction unit. Similar to our POSIT multiplication core, the adder/subtraction cores can operate well into the 5-600 MHz frequency range– between 22 and 86 MHz slower than the FLoPoCo and up-to 275 MHz slower than that of Intel's. Similar to the multiplicative case, we have many more pipeline stages compared to the IEEE-754 cores. Figure 4:d shows the ALM overhead of using POSIT addition/subtraction, which we note is more than that of the multiplier. Because floating-point addition/subtraction cannot use the hard DSP blocks, more logic has to be implemented through ALMs. Our units are up-to 3 times more area consuming than that of Intel's design. FLoPoCo remains area effective.

We acknowledge that there are many possibilities to improve our POSIT generator, especially with respect to the number of pipelines usage and their area. Finer tuning of the individual units within out generator will lead to fewer pipeline stages (and, thus, better LUT packing). We also acknowledge the possibility of having parts of the 2s-complementation done outside the core, which could partially reduce the area.

B. Linear Algebra Functions

To finalize the results, we show how we can integrate our POSIT hardware into a more productive environment for software programmers to use. We implemented three linear algebra subroutines using Intel FPGA SDK for OpenCL. The linear algebra subroutines were all using the POSIT hardware we introduced earlier in this paper. Furthermore, only basic loop pipelining and loop unrolling was performed on the kernels, leaving more advanced optimizations of such kernels for the future. The implemented routines are shown in Table I [3].

Figure 5 outlines the results from compiling our POSIT-BLAS library for the Stratix V OpenCL Board-Support Package. We see that our library consumes a generous amount of the FPGA resources, which places pressure on our implementation to not be the bottleneck of the design; for the largest 32-bit POSITs we consume nearly 70% of all ALMs and more than half of the available memory blocks. Note how the amount of area (ALMs) the library occupies decreases linearly with the used POSIT bit-width– the smaller the bit-width, the less area the design consumes. Finally, note that the

[3]An attentive reader will notice that these are the very same operators required to train deep neural networks, revealing our intentions on where to use POSITs for the future.

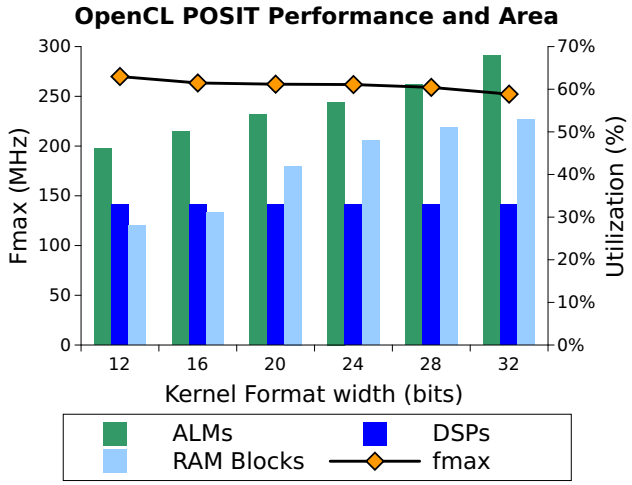

Fig. 5. Results from compiling our POSIT-BLAS (32-bit, es=3) library using Intel OpenCL for Stratix V, showing kernel frequency (timing constraints met) and resource usage of the library (bars associated with right-most y-axis).

amount of DSP blocks consumed remain identical, which is what we also saw in section VI-A.

The peak frequency (Figure 5 differs only by 25 MHz between the 32-bit and 12-bit POSIT implementation, showing the stability of our implementation with respect to how it modifies the critical path of the design. Our design keeps working at above 250 MHz, which is far beyond the frequency that the boards DDR3 memory controllers (200 MHz) operates at; this ensures us that we will likely become bandwidth- rather than compute-bound for real-world applications.

Figure 6 shows the performance we obtained with our algebra library on the three different functions. We see that `paxpy` reaches 3.37 GOPS/s (Giga-operations per second) while `pvvtmul` reach 4.19 GOPS/s. Here, both the `paxpy` and `pvvtmul` functions are memory bound and their respective arithmetic intensity do not let us get better performance– at least given our FPGA board and its external memory bandwidth (which is fully saturated for the above benchmarks). The `pgemv` application reaches 0.81 GOPS/s, which is on the lower side and more performance can be obtained by optimizing the kernel to better utilize temporal locality.

We also positioned our POSIT linear algebra subroutines against that of software emulation [20] executed on a modern Intel Xeon E5-2650v4, seen in Figure 6 (red line). We note that our FPGA platform outperforms the software implementation by several orders of magnitude. This is not surprising, since software emulation has a much larger cost than the optimized hardware we presented in this paper. However, it also shows and encourages the use of FPGAs for POSIT arithmetic to evaluate much larger problem sizes than were previous possible, allowing for better comparisons on how POSITs fare against the traditional IEEE-754.

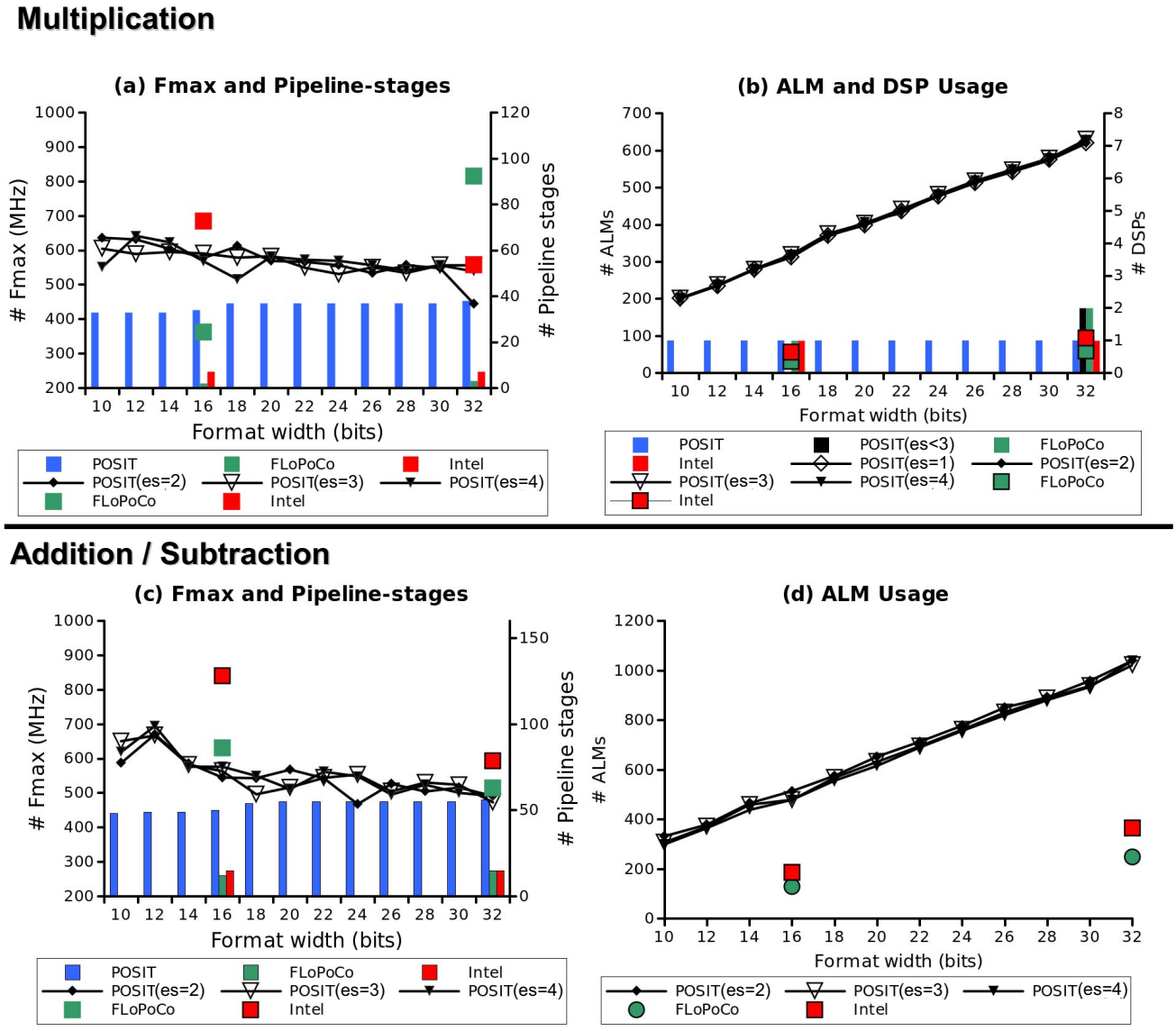

Fig. 4. Frequency, pipeline stages, area overhead (ALMs) and DSPs usage for our POSIT multiplication (a,b) and addition/subtraction (c,d) for different exponent size's (es) compared to that of FloPoCo and Intel's Megafunctions (Bars are associated with right-most y-axis).

VII. Conclusion

We have implemented, described and evaluated a prototype FPGA implementation for POSIT arithmetic. We designed a tool that automatically generates the operators based on a specification. We empirically evaluated our hardware and compared it to that of two well-known IEEE-754 implementations. Our prototype operators were capable of running at comparable frequencies, but with a significant area overhead. We further integrated our POSIT units into a more abstract programming model using Intel FPGA SDK for OpenCL by creating a library with three linear algebra functions. We evaluated these functions in terms of performance and showed the our FPGA can reach orders of magnitude better performance than currently available POSIT software libraries, allowing users to explore POSITs in more complex scenarios. Being the first prototype implementation of POSITs in hardware, we acknowledge that there is ample room for improvement, particularly in reducing the area overhead, which is where we will continue our work. We also hope to create a more complete BLAS library so that users in sciences with opportunity for changing the precision can evaluate this exciting new format.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number JP16F16764, the JSPS Postdoctoral fellowship under

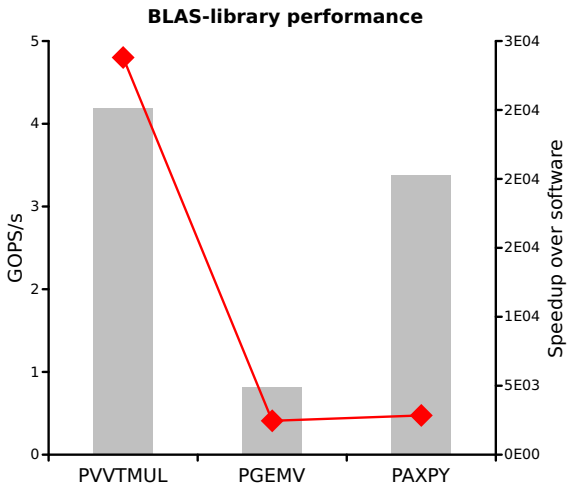

Fig. 6. Our POSIT (32-bit, 3 bit exponent) linear algebra performance in terms of obtained POSIT Giga Operations per seconds, GOPS/s (on bars associated with left y-axis), as well as the speedup over software emulation (red line, associated with right y-axis).

grant P16764, JST CREST Grant Number JPMJCR1303, and performed under the auspices of the Real-world Big-Data Computation Open Innovation Laboratory, Japan.

REFERENCES

[1] J. Dean, D. Patterson, and C. Young, "A New Golden Age in Computer Architecture: Empowering the Machine Learning Revolution," *IEEE Micro*, 2018.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ION-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[3] M. Horowitz, "1.1 Computing's Energy Problem (and what we can do about it)," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*. IEEE, 2014, pp. 10–14.

[4] "Nvidia tesla v100 gpu architecture." NVIDIA, 2017. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[5] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Hall, L. Hornof, A. Khosrowshahi *et al.*, "Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks," in *Advances in Neural Information Processing Systems*, 2017, pp. 1740–1750.

[6] "Microsoft unveils Project Brainwave for real-time AI," Microsoft, 2017. [Online]. Available: https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/

[7] E. Delaye, A. Sirasao, C. Dudha, and S. Das, "Deep Learning Challenges and Solutions with Xilinx FPGAs," in *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on*. IEEE, 2017, pp. 908–913.

[8] 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, 2017.

[9] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1, pp. 19–25, 2015.

[10] J. L. Gustafson, *The End of Error: Unum Computing*. CRC Press, 2017.

[11] F. Glaser, S. Mach, A. Rahimi, F. K. Gürkaynak, Q. Huang, and L. Benini, "An 826 MOPS, 210 uW/MHz Unum ALU in 65 nm," *arXiv preprint arXiv:1712.01021*, 2017.

[12] A. Bocco, Y. Durand, and F. De Dinechin, "Hardware support for UNUM floating point arithmetic," in *Ph. D. Research in Microelectronics and Electronics (PRIME), 2017 13th Conference on*. IEEE, 2017, pp. 93–96.

[13] J. L. Gustafson, "A radical approach to computation with real numbers," *Supercomputing frontiers and innovations*, vol. 3, no. 2, pp. 38–53, 2016.

[14] F. De Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[15] S. Bansal, H. Hsiao, T. Czajkowski, and J. H. Anderson, "High-Level Synthesis of Software-Customizable Floating-Point Cores," in *Design, Automation and Test in Europe, DATE 18 (to appear)*. IEEE, 2018.

[16] R. DiCecco, L. Sun, and P. Chow, "FPGA-Based Training of Convolutional Neural Networks With a Reduced Precision Floating-Point Library," in *International Conference on Field-Programmable Technology*. IEEE, 2017.

[17] J. Hou, Y. Zhu, Y. Shen, M. Li, Q. Wu, and H. Wu, "Enhancing Precision and Bandwidth in Cloud Computing: Implementation of a Novel Floating-Point Format on FPGA," in *Cyber Security and Cloud Computing (CSCloud), 2017 IEEE 4th International Conference on*. IEEE, 2017, pp. 310–315.

[18] ".NET implementation of POSIT," Lombiq Technologies Lt. [Online]. Available: https://github.com/Lombiq/Arithmetics

[19] "Floating-Point IP Cores User Guide," Intel, 2016.2. [Online]. Available: https://www.altera.com/documentation/eis1410764818924.html

[20] "Universal: a C++ template library for universal number arithmetic," Stillwater. [Online]. Available: https://github.com/stillwater-sc/universal