Neural Computing and Applications (2021) 33:10575–10585
https://doi.org/10.1007/s00521-021-05814-0

ORIGINAL ARTICLE

Vectorizing posit operations on RISC-V for faster deep neural networks: experiments and comparison with ARM SVE

Marco Cococcioni[1] · Federico Rossi[1] · Emanuele Ruffaldi[2] · Sergio Saponara[1]

Received: 14 October 2020 / Accepted: 5 February 2021 / Published online: 28 February 2021

Abstract
With the arrival of the open-source RISC-V processor architecture, there is the chance to rethink Deep Neural Networks (DNNs) and information representation and processing. In this work, we will exploit the following ideas: i) reduce the number of bits needed to represent the weights of the DNNs using our recent findings and implementation of the posit number system, ii) exploit RISC-V vectorization as much as possible to speed up the format encoding/decoding, the evaluation of activations functions (using only arithmetic and logic operations, exploiting approximated formulas) and the computation of core DNNs matrix-vector operations. The comparison with the well-established architecture ARM Scalable Vector Extension is natural and challenging due to its closedness and mature nature. The results show how it is possible to vectorize posit operations on RISC-V, gaining a substantial speed-up on all the operations involved. Furthermore, the experimental outcomes highlight how the new architecture can catch up, in terms of performance, with the more mature ARM architecture. Towards this end, the present study is important because it anticipates the results that we expect to achieve when we will have an open RISC-V hardware co-processor capable to operate natively with posits.

Keywords Open architectures · Vectorized operations · Posit arithmetic · Deep neural networks

1 Introduction

In the latest years, RISC-V has started to emerge as an open-source alternative CPU architecture [4, 7, 27]. Being it also royalty-free, it is the rising star competitor of Intel, AMD and ARM CPUs (both for 32 and 64-bit variants). Important software and hardware industries have endorsed and funded the project, including Intel, Microsoft and ST Microelectronics [6].

The main feature of RISC-V is its open instruction set architecture. This means that any user can extend it by adding his own instructions and functionalities: this possibility is strategic to design very low-latency co-processors and accelerators without having to treat them as external devices with memory mapping and interrupts. Furthermore, with the latest advancements of the vector extension development, RISC-V processors are able to accelerate the processing of several kernels for machine and deep learning (e.g., dot products, vector-matrix multiplications and image filtering). Lately, several real number representations have been proposed by industry and research such as Intel with Flexpoint [23, 26], Google with BFLOAT16 [8] and Facebook AI [22]. Another very promising alternative to IEEE 32-bit Floating-point standard is the posit™ number system, proposed by Gustafson [19]. This format has been proven to match single precision accuracy performance with only 16 bits used for the representation [9, 12, 16, 17, 24]. Furthermore, the first hardware implementations of this novel type are very promising in terms of energy consumption and area occupation [10, 20, 28].

In this work, we envision the adoption of these two disruptive innovations (vector extension and posit

✉ Marco Cococcioni
marco.cococcioni@unipi.it

Federico Rossi
federico.rossi@unipi.it

Emanuele Ruffaldi
emanuele.ruffaldi@mmimicro.com

Sergio Saponara
sergio.saponara@unipi.it

[1] Department of Information Engineering, University of Pisa, 56122 Pisa, Italy

[2] MMI spa, Pisa, Italy

arithmetic) within the same architecture. Our ultimate goal is to extend RISC-V to be able to use a Posit Processing Unit (PPU) as a co-processor, by extending the processor instruction set architecture (ISA). While going towards this end, we can anyway gain great benefits from the posit format. It is of particular interest knowing the potential benefits of posit-based co-processor in a killer application such as Deep Neural Networks.

In this work, we assess the quality of the vectorization of RISC-V operations when using posit numbers and we compare it with ARM SVE. The paper is organized in the following way. In Sect. 2, we briefly present an overview of the posit format, along with recent improvements and findings achieved at University of Pisa, in collaboration with MMI spa, on information processing using posit numbers. In Sect. 3, we summarise the core aspects of the RISC-V architecture. In particular, we focus on the RISC-V vector extension, showing the principal component used in the rest of the work. In Sect. 4, we present the implementation of vectorized posit operations inside the cppPosit library (a C++ Posit library developed and maintained by the authors) for RISC-V, following the same approach of our previous implementation of the ARM SVE vectorized operations [14]. In particular, we focus on the implementation of posit encoding and decoding from/to the floating-point format. In Sect. 5, we illustrate the benchmarks used to test our implementation. The tests represent different core operations of DNNs, including convolutions, dot-products and matrix-matrix multiplications as well as activation functions. In particular, the proposed activation functions are fast approximated versions of the real ones; these functions can be computed just by an arithmetic-logic unit, thus being highly vectorizable. In Sect. 6, we outline our vision to enable hardware accelerators for posit operations for a RISC-V processor focusing on the latency and implementation complexity of the different solutions. Finally, we also present a comparison with ARM SVE and future works. In Sect. 7, we draw some conclusions.

2 Posit arithmetic

The posit format [11, 12, 16, 19] is a configurable fixed length format for real number representation; the format configuration involves the number of overall bits (*nbits*) and the maximum number of exponent bits (*esbits*).

2.1 Format overview

As shown in Fig. 1, a posit number is composed by a maximum of 4 fields: i) sign (1-bit), ii) regime (variable length), iii) exponent (maximum of *esbits*) and iv) fraction (variable length). Note that posits are encoded using 2's complement. The regime field is a particular one; its length is identified by a series of bit equal to 1 (or 0) terminated by a stop-bit with the opposite value. The value of the regime field is then the number of equal bits in the so discovered bit-string.

Expression (1) shows the mathematical relation between the posit bit content (v) and the represented value (x). k is the regime value computed as described before and $useed = 2^{2^{esbits}}$. e and f are, respectively, the exponent and fraction values decoded from the base-2 representation (Fig. 2).

$$x = \begin{cases} 0, \text{ if } \quad \text{v} = 0 \\ \text{NaN, if } \quad \text{v} = -2^{(nbits-1)} \\ sign(v) \times useed^{k} \cdot 2^{e} \cdot (1+f), \text{ otherwise} \end{cases} \tag{1}$$

2.2 Advantages over IEEE 32-bit Floats

As widely described in [18, 19] several issues are afflicting the IEEE Float32 format that are addressed by this novel format:

- Waste of bit patterns: IEEE Float32 wastes millions of patterns for NaN values;
- Mathematically incorrect: two representation of 0 (± 0);
- Non-configurable accuracy: pre-determined number of exponent and mantissa bits;
- Being a more nonlinear and compressed representation, it allows more powerful bit manipulations (changing the bit string of a posit in an appropriate way, using the ALU alone, can lead to interesting nonlinear transformations of the posit number itself).

The application of posit numbers to Deep Neural Networks (DNN) has been independently proven to this authors and others, to perform as good as float numbers [15, 25] with half the bits (or even less), as reported in Table 1. The table reports the comparison between different configurations of posit with a 32-bit float, on the German Traffic Sign Recognition Benchmark (GTRSB) dataset. In the Table 1, a 10-bit posit arithmetic allows for the same detection and classification accuracy of 32-bit float, while

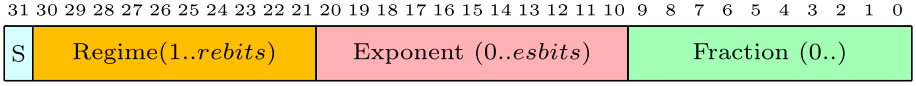

Fig. 1 Example of a Posit$<32, 11>$ configuration, where $nbits = 32$ and $esbits = 11$

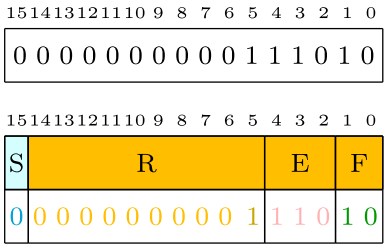

Fig. 2 An example of posit instance with 16 bits and 3 exponent bits. The associated real value to the shown posit is: $+256^{-9} \cdot 2^{6} \cdot (1 + 2/4) = 2.0329 \cdot 10^{-20}$.

Table 1 Comparing the classification accuracy of a DNN on the GTRSB dataset, when using different real number representations and different number of bits

Type	Acc. (%)
Float32	94.01
posit $<16, 0>$	94.02
posit $<14, 0>$	94.01
posit $<12, 0>$	94.01
posit $<10, 0>$	94.01
posit $<8, 0>$	93.81

the accuracy reduction of a 8-bit posit is limited to 0.2% (for a data size saving of a factor 4 vs 32-bit float).

2.3 No exponent bit case

As shown by the authors in [13], the posit format gains interesting properties when configured with $esbits = 0$. This particular configuration allows the implementation of fast versions of common operations (a little approximation is introduced in some of them); the new versions can be computed just by using the arithmetic-logic unit (ALU) of the CPU since they only involve bit manipulation and integer operations. Among the basic operations that can be accelerated this way, there are the double and half operators ($2x$ and $x/2$), the inverse operator ($1/x$) and the one's complement ($1 - x$).

Furthermore, some common use activation functions for Deep Neural Networks (DNNs) can be implemented this way. The basic building block is represented by the fast approximation of the Sigmoid function (from [19]). Starting from this one, we developed the others as a simple combination of the previous operators and the Sigmoid.

What we obtained is the fast approximation of the hyperbolic tangent (FastTanh [11, 13]):

$$\tanh(x) = 2\text{sigmoid}(2x) - 1 = -(1 - 2\text{sigmoid}(2x))$$

and the fast approximation of the extended linear unit (FastELU [13]):

$$e^{x} - 1 = -2 \cdot \left[1 - \frac{1}{2\text{sigmoid}(-x)}\right]$$

The possibility to implement several operations as simple ALU instructions leads to some interesting aspects:

- It allows the ALU emulation of posit operations without specific hardware support.
- It allows an operator to be implemented as a sequence of ALU instructions; this means that every operation implemented this way can be easily vectorizable.

2.4 Past achievements concerning posit-based DNNs

In previous work, we have been able to:

- develop our posit software library (cppPosit, see Sect. 4);
- implement fast approximated activation functions, only possible when using posits, which exploit the ALU (no PPU necessary for it): [13, 15];
- fast matrix-vector multiplication, thanks to vectorization (demonstrated on ARM CPUs, using SVE: [14]).

In this paper, we aim the present what we did to obtain the same achievements, but this time on an open processor, the RISC-V.

3 The RISC-V architecture

The RISC-V [4] architecture is a modular, open-source and royalty-free instruction set architecture (ISA) and comprises both 32 and 64-bit flavours. The overall ISA is composed of smaller sub-ISAs among which there are the base subsets. These subsets are referred as *base integer instruction sets* and identified by the letter `I`. Besides, a RISC-V-based architecture can present some other extensions; some of them are referred to as *frozen*. This means that their encoding and behaviour has been ratified and will not change during the current draft of the architecture. These extensions include integer multiplication/division operations (`M`), single (`F`), double (`D`) precision floating point operations (following the IEEE 754 Float standard) and atomic instructions (`A`).

3.1 The RISC-V vector extension

A very interesting and under development extension is the vector one (`V`). This extension aims to provide single-instruction multiple-data (SIMD) capabilities to the RISC-V architecture. By design, this extension can seamlessly exploit either the CPU registers or a special vector co-processor for hardware acceleration. Any RISC-V-based

architecture implementing this extension will define some parameters:

- Number of vector registers (standard is 32)
- *vlen*: size (in bits) of the vector registers (e.g., 256)
- *elen*: maximum supported size for a single element (e.g., 64 for a 64-bit integer or double)

The idea behind the vector extension is the same of the ARM scalable vector extension (SVE) architecture [1]; there is not a predetermined vector length (as happens in the Intel SIMD extensions) but a special instruction `vsetvl`. This instruction takes as input a requested vector length *vreql* and returns the granted vector length *vgrant* as in next expression:

$$vgrant = \min(vlen, vreql)$$

This design allows porting an application between RISC-V architectures, without re-writing a single line of code and, in case of furtherly compatible architectures, without recompilation. Moreover, this will help us later when simulating the same program with different vector configurations.

4 The cppPosit library

The support for posit arithmetic is offered by the cppPosit library. This library has been developed in Pisa, and it is maintained by the authors of this work. The library uses advanced templatization techniques from C++14 to ease the definition of posit configurations at compile time. Posit operations are classified into four different levels ($\mathcal{L}1$-$\mathcal{L}4$) with increasing computational complexity [13]. The simplest and fastest level is called $\mathcal{L}1$ and comprises all the operators described in Sect. 2.3. The library also offers three back-ends to rely on for posit operations that cannot be emulated via ALU:

- Floating point back-end, using the standard FPU support for operations;
- Fixed point back-end, exploiting big-integer support (64 or 128 bits) for operations;
- Tabulated back-end, generating lookup tables for most of the operations (suitable for Posit$<[8, 12], * >$ due to table sizes).

4.1 Posits and RISC-V vectorization

Vectorization of posit operations was already proved to be interesting in the ARM SVE environment in [14]. As shown in that work, each function aimed at vectorizing posit operations has three main parts: i) prologue, ii) body, iii) epilogue.

In the prologue, we need to prepare the vector containing posit data referring to the underlying vector architecture. This phase varies whether we are implementing $\mathcal{L}1$ function or not. In the former case, the prologue is just a `reinterpret_cast` from the posit type to the underlying integer holder type. Instead, when dealing with non-$\mathcal{L}1$ function, this phase is also devoted to the conversion from posit type to a suitable back-end type (e.g., for RISC-V we choose to convert it to IEEE Float32). In the body, there is the actual implementation of the vectorized operation. Finally, in the epilogue, we need to build back the posit vector (the same considerations of the prologue hold here).

An important focus must be put on the epilogue and prologue phases: since these phases employ posits with an overall size of 16 or 8, we are performing a data compression by a factor 2 or 4. Moreover, even if we use 32-bit floats to perform computations, the data expansion is performed only inside the vector processing unit. Therefore, we are transferring compressed data from the scalar CPU registers to the vector ones. We can thus transfer from twice to quadruple the data with a single vector load instruction. This means that just by using posits as a compressed information storage can lead to great optimizations in data transfer. Figure 3 shows the overall idea behind posit vectorization in cppPosit.

5 Experimental results

In this section, we will present the benchmarks for the novel RISC-V-"V" vectorized approach on several benchmark kernels. Instead of relating our benchmarks to a particular DNN library or implementation, we decided to present the benchmarks on simple and core building blocks for DNNs.

Firstly, we will present benchmarks of vectorized $\mathcal{L}1$ activation functions such as sigmoid, hyperbolic tangent and extended linear unit. Secondly, we will present benchmarks on matrix and vector operations such as dot product, convolution and general matrix-matrix multiplication.

The RISC-V benchmark binaries were generated using the Barcellona Supercomputing Centre (BSC) LLVM cross compiler (`clang++ 11.0`). As for now, this is the only compiler providing high-level intrinsics for the RISC-V vector extension [3]. The RISC-V binaries were then executed on the RISC-V Spike simulator (`riscv-isa-sim RVV version 0.8`) [5].

The ARM benchmarks binaries were generated using the upstream branch of the GCC compiler (`GCC 10.0`) with SVE intrinsic support. The examples were executed on a static, user-space QEMU (`QEMU 5.0`) installation for ARMv8.2 that supports SVE instructions.

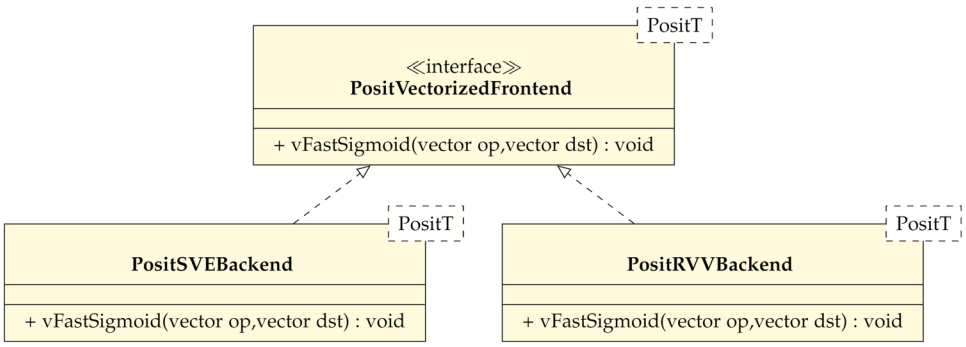

Fig. 3 UML class diagram for the overall implementation of vectorized operations on posits. Both ARM SVE (left) and RISC-V (right) vectorized operations are supported by our cppPosit library

All the simulations were executed on a 8 core Intel(R) Core(TM) i7-9700 CPU with 3.00GHz base frequency.

For each benchmark result, we analyzed the timing performance of prologue, body and epilogue to address the overhead that is introduced by the first and the last part (as already discussed in Sect. 4.1. Moreover, we compared the vectorized performance varying the vector length against the non-vectorized implementation (called *naive* from now on).

5.1 Vectorization of posit encoding and decoding

Since we could not rely on auto-vectorization due to compiler limitations there was the need to provide a vectorized implementation of posit decoding and decoding operations for prologue and epilogue blocks.

Completely decoding a posit means taking its representing integer and extract the four fields described in Sect. 2.1. This can be accomplished using just arithmetic and logic integer operations. Algorithm 1 shows the steps we used to unpack a posit into its (at most) four fields. Given these fields, we can build a float number.

Let F be a IEEE754 FP32 number:

$$F = <sign(s_F, 1), exponent(es_F, 8), fraction(m_F, 23) >$$

The represented value (see (1)) is:

$$x_F = (-1)^{s_F} \cdot 2^{es_F - 127} \cdot (1 + f_F / 2^{23})$$

Let P be a posit $<nbits, esbits>$:

$$P = <(s_P, 1), (k_P, regbits), (es_P, es), (m_P, fracbits) >$$

where $es \leq esbits$.

The represented real value is:

$$x_P = (-1)^{s_P} \cdot 2^{(k_P \cdot 2^{esbits})} \cdot 2^{es_P} \cdot (1 + f_P / 2^{fracbits})$$

where $fracbits = nbits - 1 - regbits - es$.

For the conversion, we want that $x_P = x_F$. This implies that:

$$s_P = s_F$$

$$es_F = k_P \cdot 2^{esbits} + es_P + 127 \quad (2)$$

$$m_F = m_P \cdot 2^{23 - fracbits} \quad (3)$$

Algorithm 1 Posit decoding algorithm: *signBit* is the posit most significant bit, *extraBits* takes into account of underlying holder type that may not be aligned with the posit size (e.g. Posit$\langle 10, x\rangle$ stored in an $int16\_t$ type), *positHolderMSB* is the holder type most significant bit. The *findLeftMostSet* function is used to find the index of the first set bit starting from the most significant bit. It is commonly known as *count leading zeroes* (CLZ). The *getBitSetLeft(bitstring,n)* is used to extract n bits from *bitstring* starting from the most significant one.

Input: positRepresentation
Output: sign,regime,exponent,fraction
1: **sign** = (positRepresentation & signBit) != 0
2: pa = sign ? -positRepresentation : positRepresentation
3: pars = pa ≪ (extraBits+1)
4: stop = (pars & positHolderMsb) != 0
5: index = stop ? findLeftMostSet(∼pars) : findLeftMostSet(pars)
6: **regime** = stop ? index - 1 : -index
7: regimeLength = index + 1
8: pars = pars ≪ regimeLength
9: **exponent** = getBitSetLeft(pars,esbits)
10: **fraction** = pars ≪ esbits ▷ Fraction in MSBs

Encoding a posit means taking any real number representation (e.g., IEEE Float32) and computing the integer that represents the posit as described in Sect. 2.1. This can be accomplished using just arithmetic and logic integer operations. Algorithm 2 shows the steps we used to unpack a posit into its (at most) four fields. Once we decode the floating point into its 3 fields we can reason on how to retrieve posit fields. If we look at Eq. (2), we can see that k_P is the result of an integer division between es_F and 2^{esbits} and $es_P + 127$ is the remainder of the integer division. This means that we can retrieve the two fields as:

$$k_P = es_F / 2^{esbits}$$

$$es_P = es_F - k_P \cdot 2^{esbits}$$

Now, we consider the fraction part. As shown in Eq. (3), retrieving m_P is equivalent to a logical right shift of $23 - fracbits$ on m_F, thus obtaining the *fracbits* most significant bits.

5.2 Vectorized activation function benchmarks

In this subsection, we report in Fig. 4 and 5 the results of the benchmarks concerning the vectorization of the activation functions (Sigmoid and ELU, respectively). These benchmarks consisted in the execution of the vectorized activation functions on random vectors of 8192 items varying the vector length and using Posit$<16,0>$ and Posit$<8,0>$.

These results are particularly groundbreaking; we are indeed reducing the processing time of an activation function by a factor 10 at least. This is easily obtained by just applying posit properties. Moreover, there is nothing similar that can be obtained with the floating point format to achieve similar speedup while varying vector length and data size.

This is extremely important, since modern neural network architectures can require the computation of nonlinear activation functions like ELU on vectors with up to tens of thousans of elements.

Algorithm 2 Float to posit conversion algorithm. The *pack* function at line 11 is used to build the posit representing integer using the field built in the algorithm: if the sign is negative, we firstly compute the posit for the positive value than we apply the 2's complement to the posit fields to change sign.

Input: floatRepresentation
Output: positRepresentation
1: sign = (floatRepresentation & floatSignBit) != 0
2: fExponent = (floatRepresentation $\ll$ 1) $\gg$ 24
3: normFExponent = fExponent - 127
4: fFraction = (floatRepresentation $\ll$ 9) ▷ Fraction in MSBs
5: pRegime = normFExponent/esbits
6: pRegimeBits = pRegime + 1
7: expBits = min(nbits - pRegimeBits - 1,esbits)
8: fracBits = nbits - pRegimeBits - 1 - expBits
9: pExp = normFExponent - pRegime
10: pFrac = fFraction $\gg$ (nbits - fracbits)
11: positRepresentation = pack(sign,pRegime,pExp,pFrac)

We implemented both algorithms using hand-vectorization for the RISC-V platform, using the "V" extension intrinsics. The *findLeftMostSet* is particularly interesting, since there was not a native vectorized implementation in the RISC-V "V" extension at the moment of the development. Algorithm 3 shows our choice for the implementation. We implemented also this algorithm using the RISC-V vectorization intrinsics.

5.3 Vectorized matrix-vector operation benchmarks

In this subsection, we show the results concerning vectorized matrix-vector operation benchmarks. The benchmark parameters are the following:

- Dot product: timing performance on a dot product between vectors of size 64.

Algorithm 3 Count Leading Zeros (CLZ) function implemented using bit manipulation only: an example for a 16-bit integer

Input: unsignedInt16
Output: firstSet
1: firstSet = (bitString > 0xFF) $\ll$ 3; bitstring $\gg$ firstSet
2: q = (bitString > 0xF) $\ll$ 2; bitstring $\gg$ q
3: firstSet = firstSet | q
4: q = (bitString > 0x3) $\ll$ 1; bitstring $\gg$ q
5: firstSet = firstSet | q
6: firstSet = firstSet | (bitString $\gg$ 1)

- Convolution: timing performance on a 3×3 convolution over matrices of size 64×64.
- Matrix-matrix multiplication (GEMM): timing performance on a matrix multiplication between matrices of size 32×32.

For each benchmark, we report two different types of result:

- Performance comparison (Figs. 6, 7, 8): timing performance comparison varying the vector length and disabling vectorization. We used a logarithmic scale to represent these results in order to increase plot readability.
- posit decoding/encoding overhead evaluation (Figs. 9, 10, 11): timing performances are decomposed to evaluate the contribution of each part of the algorithm.

5.4 Analysis of results and discussions

In this section, we will analyse and discuss the results shown before.

Firstly, as reported in Figs. 4 and 5, the speed-up gained by the vectorization of activation functions is impressive in both cases, with vectorized performance being one order of magnitude smaller than the non-vectorized ones. Moreover, it is evident how much the algorithm benefits from increasing the vector length and halving the data size. Note that this speed-up could not be achieved using the IEEE Float32 format; in fact we could apply the series of arithmetic and logic operations thanks to the posit representation with zero exponent bits.

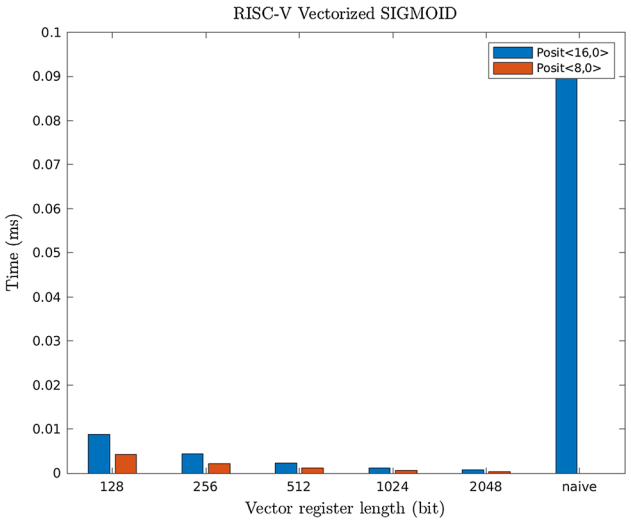

Fig. 4 Comparing the elapsed time when computing the approximated Sigmoid activation function for different vector register lengths. The comparison is provided both for posit $< 8, 0 >$ and posit $< 16, 0 >$. Naive stands for "unvectorized"

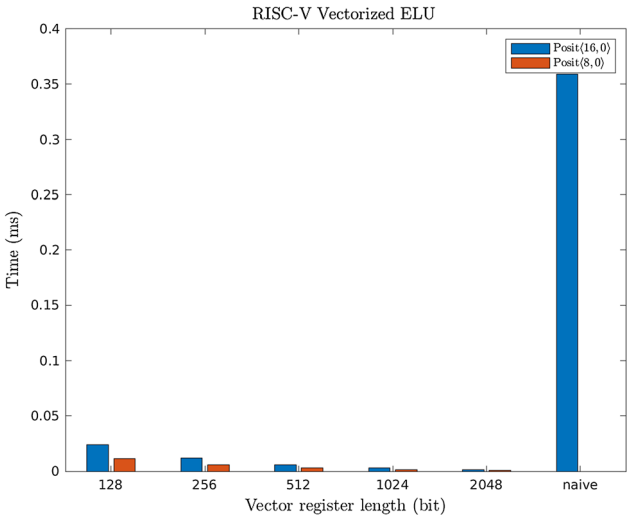

Fig. 5 Comparing the elapsed time when computing the approximated ELU activation function for different vector register lengths. The comparison is provided both for posit $< 8, 0 >$ and posit $< 16, 0 >$

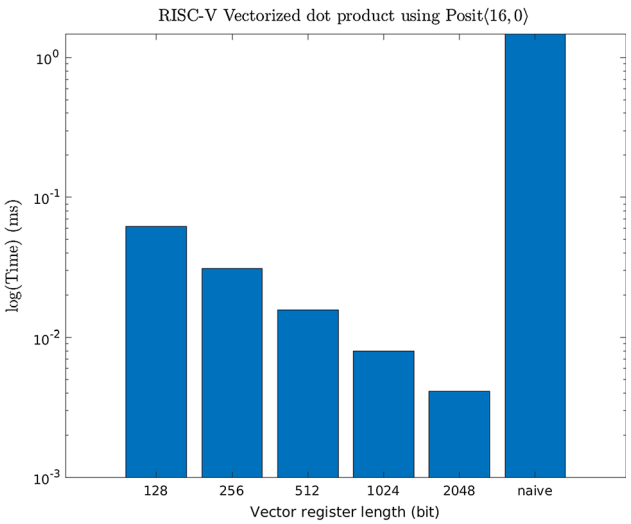

Fig. 6 Dot product time comparison for different vector register lengths (posit $< 16, 0 >$ has been used here). Y axis is reported in logarithmic scale for readability

Secondly, speaking of matrix-vector operations, again the speed-up when using vectorized algorithms is massive, reducing processing time by more than 2 orders of magnitude (note the logarithmic scale of the plots in Figs. 5, 6 and 7).

Thirdly, the same benchmarks were executed on the ARM environment explained at the beginning of this section. In particular, we considered the 512-bit vector length for the comparison to match the first hardware ARM SVE platform on the market (the Fujitsu A64FX processor [2]). As shown in Fig. 12, if we compare the vectorization on the

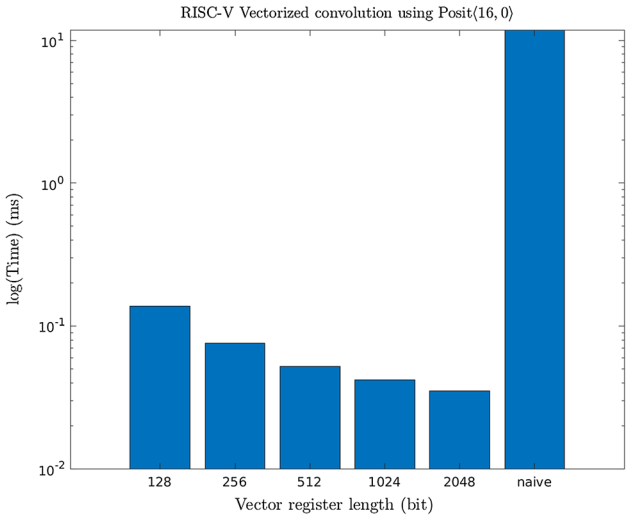

Fig. 7 Convolution performance comparison on posit$<16,0>$. Y axis is reported in logarithmic scale for readability

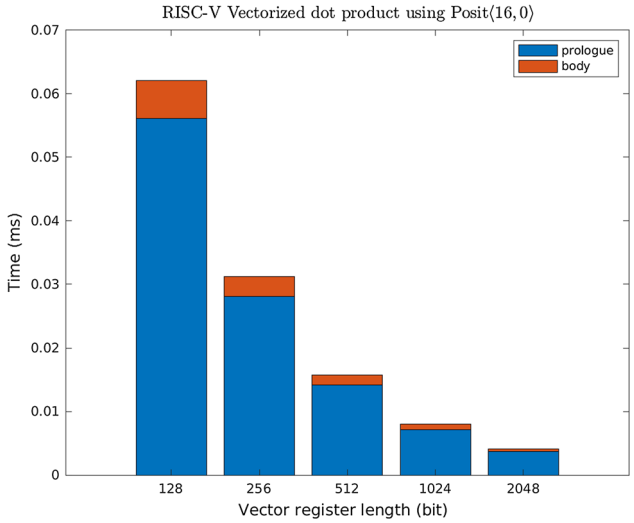

Fig. 9 Dot product performance comparison on posit$<16,0>$ with separation of the operation blocks. This picture provides additional details to Fig. 6. Here, the epilogue time is not reported, since the time taken to transform a single float into a posit is negligible

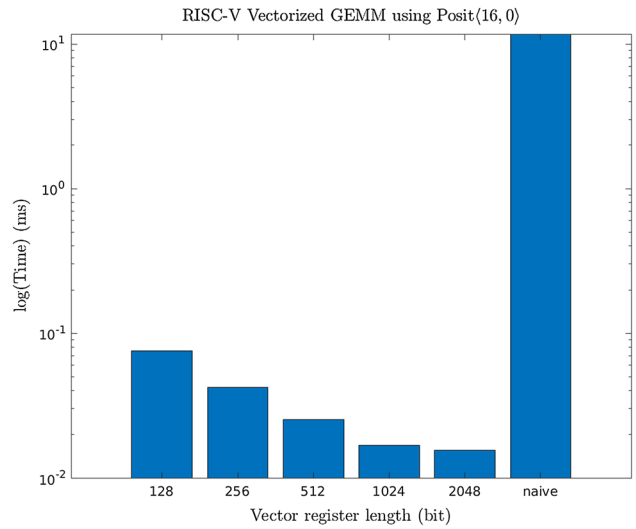

Fig. 8 GEMM performance comparison on posit$<16,0>$. Y axis is reported in logarithmic scale for readability

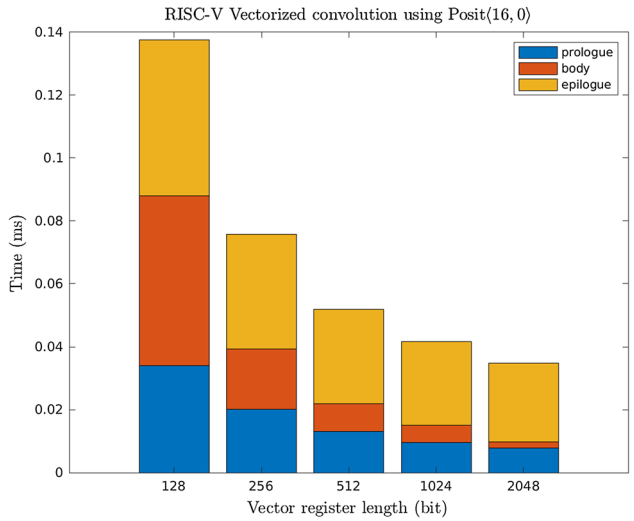

Fig. 10 Convolution product performance comparison on posit$<16,0>$ with separation of the operation blocks. This picture provides additional details to Fig. 7

RISC-V platform to the ARM SVE platform, the results are comparable, basically resulting in a draw.

Finally, the newly proposed approach for decoding and encoding vectorization of posits brought a substantial speed-up in the prologue and epilogue phases. As we can see from Figs. 9, 10 and 11 the vectorized phases benefit from increasing the vector register lengths. However, there is still some overhead that afflicts the epilogue part due to costly conversions between 32-bit integers (for Float representation) and posits holder integers (16-bit integers in this case).

6 Future work

Having a dedicated posit processing unit is critical to increase the architecture performance removing the software emulation bottleneck. There are three main ways to equip a RISC-V CPU with a hardware PPU:

1. include the PPU within the processor [21]. This requires the introduction of an additional instruction set and the instrumentation of the compiler.
2. use the PPU as a slave peripheral. The peripheral can be an external FPGA board connected via PCI Express

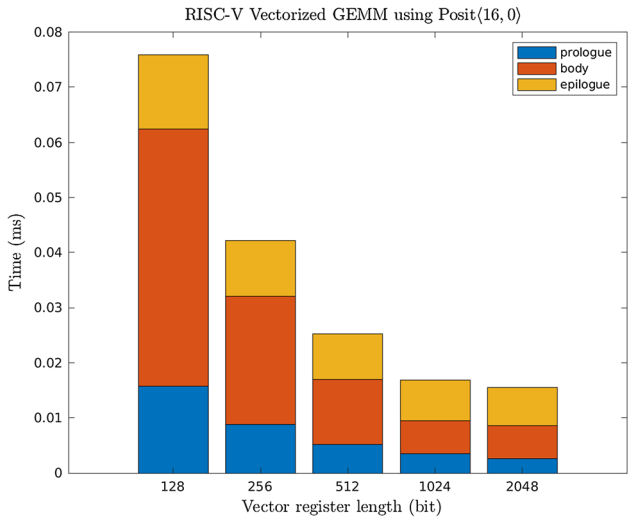

Fig. 11 GEMM product performance comparison on posit $<16,0>$ with separation of the operation blocks. Therefore this picture is related to Fig. 8, but without reporting the time for the naive approach, of course

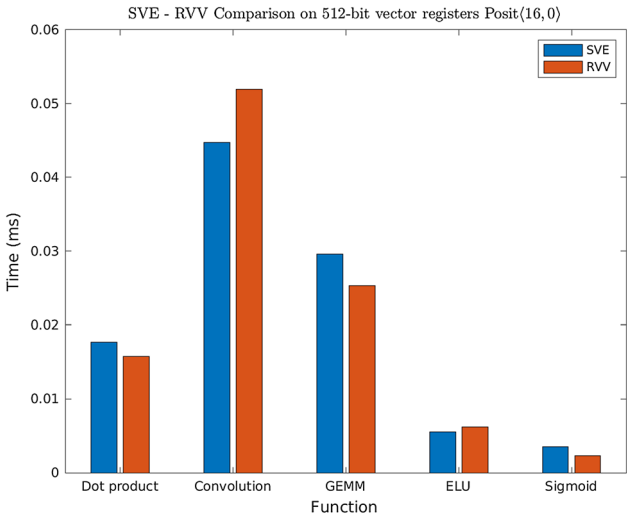

Fig. 12 Comparison between SVE and RISC-V using Posit $<16,0>$ with 512-bit vector registers

bus. This is similar to how GPUs are connected to CPUs nowadays. This solution has the highest latency, since we need to communicate with an external peripheral via bus communication.

3. design the PPU as an IP-core to be included within the processor chip. This can be viewed as a co-processor approach, thus having PPU and CPU on the same SoC (system-on-chip). In terms of latency, this solution is an intermediate one.

We are currently working to implement a PPU for RISC-V following option 1. However, even without a hardware PPU, the solution proposed in this paper (vectorized posit operations, emulated on ALU of FPU) is still of interest in particular situations, as discussed below.

Figure 13 shows some computing environments, with increasing computing power, cost and energy consumption: i) micro-controllers, ii) CPUs without the FPU, iii) CPUs with the FPU, iv) Many Core CPUs without GPUs, and v) Multi (or Many) Core CPUs with GPUs. With "Many Core" CPU (MaCPU), we designate a CPU having more than 100 cores, while with "Multi Core" CPUs we indicate those having up to 100 cores.

The solution proposed in this work is of interests for cases ii), iii) and iv). Indeed, In case ii) the use of ALU-emulated Posit-DNNs is particularly interesting and justified (see [13, 15]). Also in case iii), the approach proposed here is a viable and appealing solution, provided that the CPU supports vectorization. Case iv) is the situation where the solution proposed here is expected to obtain the best speedup, since we can exploit the massive data and instruction parallelism introduced by many core architectures to increase the DNN processing capabilities.

In all the cases, the use of posit numbers can at least halve (when using posit $<16,x>$) the representation size, thus doubling the bandwidth of the information and improving the usage of the cache. Moreover, reducing the information size is extremely useful when combined with the vector engines. With a half of the element size, we can fit double the elements inside a vector register, as demonstrated in [14] for ARM CPUs and here for RISC-V CPUs. Finally, when posit $<8,x>$-based DNN reaches a satisfying accuracy, the benefit with respect to 32-bit floats is much more impactful.

7 Conclusions

In this paper, we presented the implementation of posit vector operations for DNNs using the RISC-V open-source hardware platform. We proposed an extension of our cppPosit library enabling dot-product, matrix-matrix and convolution operations in the RISC-V Vector extension environment. Moreover, we provided the implementation of fast approximated activation functions only using vectorized integer arithmetic and logic. As reported in the experimental results, we gained a significant speed-up from the hand-vectorization of posit operations, including the encoding and decoding of the novel format to the standard IEEE 32-bit floating-point format. Furthermore, we managed to catch up the ARM SVE results when vectorizing the same operations. The promising results may indicate that open-source hardware platform like RISC-V, along with open-source DNN software implementations, may

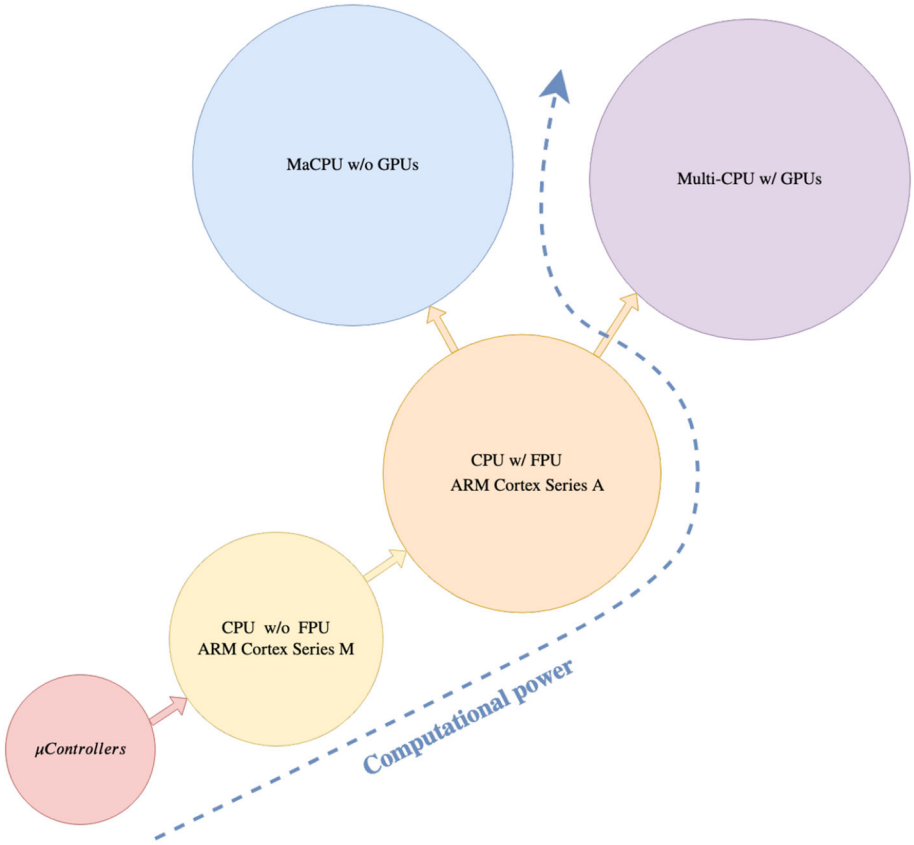

Fig. 13 Representation of different computing environment, with increasing computational power. In some configurations above, the FPU can be absent (to save energy and/or production costs as with the ARM Cortex Series M). So that the use of posit with an ALU backend is an attractive solution. There are also situations where the FPU is present but the GPUs are absent: again, in such situations (e.g., ARM Cortex Series A), the use of ALU-emulated (or even FPU-emulated) posits is a viable solution to speedup DNNs or save memory. In some situations, the user can have a CPU with a high number of cores (they are more and more affordable, today—the AMD EPYC 7000-Series, for instance, can have 128 cores) and no GPUs: even this one is an interesting case where ALU/FPU-emulated posits can speedup DNNs. Of course, having a hardware PPU provides higher performance levels and would allow the use of DNNs of all the types, without restrictions, as discussed in Sect. 6. In the picture, MaCPU stands for many-core CPU, i.e., a CPU with more than 100 cores

enable a brand new class of completely open DNN computing environments.

Acknowledgements The present study has been possible thanks to the support of the team of the Barcelona Supercomputing Center dedicated to RISC-V extension. We have conducted this work with their invaluable support, as a common partner of the same H2020 EPI project. To them goes our most sincere appreciation.

Funding Open access funding provided by Università di Pisa within the CRUI-CARE Agreement..

Compliance with ethical standards

Conflicts of interest The authors declare that they have no conflict of interest.

References

1. ARM HPC tools for SVE. https://developer.arm.com/tools-and-software/server-and-hpc/arm-architecture-tools/documentation/introducing-scalable-vector-extension-sve. Accessed 7 July 2020
2. Post-K Supercomputer with Fujitsu's Original CPU, A64FX Powered by ARM ISA. (2019) https://www.fujitsu.com/global/Images/post-k_supercomputer_with_fujitsu's_original_cpu_a64fx_powered_by_arm_isa.pdf. Accessed 7 July 2020
3. V for vector: software exploration of the vector extension of risc-v. (2020) https://www.european-processor-initiative.eu/v-for-vector-software-exploration-of-the-vector-extension-of-risc-v/. Accessed 7 July 2020

4. RISC-V ISA Specification. https://riscv.org/specifications/isa-spec-pdf/. Accessed 11 March 2020
5. Spike, a RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim. Accessed 7 July 2020
6. RISC-V History. https://riscv.org/risc-v-history/. Accessed 28 May 2020
7. Asanović K, Patterson DA.:(2014) Instruction sets should be free: The case for RISC-V. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146
8. Burgess N, Milanovic J, Stephens N, Monachopoulos K, Mansell D. (2019): Bfloat16 processing for neural networks. In: 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), pp. 88–91. https://doi.org/10.1109/ARITH.2019.00022
9. Carmichael Z, Langroudi HF, Khazanov C, Lillie J, Gustafson JL, Kudithipudi D (2019) Deep positron: A deep neural network using the posit number system. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pp 1421–1426
10. Chaurasiya R, Gustafson J, Shrestha R, Neudorfer J, Nambiar S, Niyogi K, Merchant F, Leupers R (2018) Parameterized posit arithmetic hardware generator. In: 2018 IEEE 36th International Conference on Computer Design (ICCD), pp. 334–341. https://doi.org/10.1109/ICCD.2018.00057
11. Cococcioni M, Rossi F, Ruffaldi E, Saponara S (2019) A fast approximation of the hyperbolic tangent when using posit numbers and its application to deep neural networks. In: Int. Workshop on Applic. in Electronics Pervading Ind., Envir. and Society (ApplePies'19) https://doi.org/10.1007/978-3-030-37277-4_25
12. Cococcioni M, Rossi F, Ruffaldi E, Saponara S (2019) Novel arithmetics to accelerate machine learning classifiers in autonomous driving applications. In: In Proc. of the 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS'19), pp. 779–782. https://doi.org/10.1109/ICECS46596.2019.8965031
13. Cococcioni M, Rossi F, Ruffaldi E, Saponara S (2020) Fast approximations of activation functions in deep neural networks when using posit arithmetic. Sensors **20**(5) https://www.mdpi.com/1424-8220/20/5/1515
14. Cococcioni M, Rossi F, Ruffaldi E, Saponara S (2020) Fast deep neural networks for image processing using posits and ARM Scalable Vector Extension. Journal of Real-Time Image Processing pp. 1–13. https://doi.org/10.1007/s11554-020-00984-x
15. Cococcioni M, Rossi F, Ruffaldi E, Saponara S (2021) Novel arithmetics in deep neural networks signal processing for autonomous driving: Challenges and opportunities. IEEE Signal Processing Magazine, Special Issue on Autonomous Driving, vol. 38, no. 1, pp. 97–110. https://doi.org/10.1109/MSP.2020.2988436
16. Cococcioni M, Ruffaldi E, Saponara S (2018) Exploiting posit arithmetic for deep neural networks in autonomous driving applications. In: In Proc. of the 2018 IEEE International Conference of Electrical and Electronic Technologies for Automotive (Automotive'18), pp. 1–6. https://doi.org/10.23919/EETA.2018.8493233
17. Fatemi Langroudi SH, Carmichael Z, Gustafson J, Kudithipudi D (2019) Positnn framework: Tapered precision deep learning inference for the edge. pp. 53–59. https://doi.org/10.1109/SpaceComp.2019.00011
18. Gustafson JL (2015) The end of error: unum computing. Chapman and Hall/CRC, Cambridge
19. Gustafson JL, Yonemoto IT (2017) Beating floating point at its own game: posit arithmetic. Supercomput Front Innov 4(2):71–86
20. Jaiswal MK, So HKH. (2018): Universal number posit arithmetic generator on FPGA. In: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), pp 1159–1162. https://doi.org/10.23919/DATE.2018.8342187
21. Jaiswal MK, So HKH (2019) PACoGen: A hardware posit arithmetic core generator. IEEE Access 7:74586–74601
22. Johnson J (2018) Rethinking floating point for deep learning. CoRR. http://arxiv.org/abs/1811.01721. Accessed 7 July 2020
23. Köster U, Webb T, Wang X, Nassar M, Bansal AK, Constable W, Elibol O, Gray S, Hall S, Hornof L. et al. (2017): Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In: In Proceedings of teh 31st Conference on Neural Information Processing Systems (NIPS'17), pp. 1742–1752
24. Lu J, Fang C, Xu M, Lin J, Wang Z (2020) Evaluations on deep neural networks training using posit number system. IEEE Transactions on Computers, pp 0–1
25. Murillo R, Del Barrio AA, Botella G (2020) Deep pensieve: a deep learning framework based on the posit number system. Digital Signal Process 102:102762
26. Popescu V, Nassar M, Wang X, Tumer E, Webb T. (2018): Flexpoint: Predictive numerics for deep learning. In: Proceedings of the 25th IEEE Symposium on Computer Arithmetic (ARITH'18), pp 1–4. https://doi.org/10.1109/ARITH.2018.8464801
27. Waterman A, Lee Y, Patterson DA, Asanovic K. (2011): The RISC-V instruction set manual, volume I: Base user-level ISA. EECS Department, UC Berkeley, Technical Report UCB/EECS-2011-62 **116**
28. Zhang H, Ko S (2020) Design of power efficient posit multiplier. IEEE Trans Circuits Syst II Express Briefs 67(5):861–865