Novel Arithmetics to Accelerate Machine Learning Classifiers in Autonomous Driving Applications

Marco Cococcioni[*], *IEEE SM*, Federico Rossi[*], Emanuele Ruffaldi[#], Sergio Saponara[*], *IEEE SM*
[*]Dept. of Information Engineering, University of Pisa, Italy - [#]MMI spa, Calci, Pisa, Italy

Abstract. Autonomous driving techniques frequently need the clustering and the classification of data coming from several input sensors, like cameras, radar and lidars. These sub-tasks need to be implemented0020in real-time in embedded on-board computing units. The trend for data classification and clustering in the signal processing community is moving towards machine learning (ML) algorithms. One of them, which plays a central role, is the *k*-nearest neighbors (*k*-NN) algorithm. To meet stringent requirements in terms of real-time computing capability and circuit/memory complexity, ML accelerators are needed. Innovation is required in terms of computing arithmetic since classic integer numbers lead to low classification accuracy with respect to the needs of safety critical applications like autonomous driving. Instead, floating numbers require too much circuit and memory. To overcome these issues the paper shows that the use of a new format, called Posit, implemented in a new *cppPosit* software library, can lead to a *k*-NN implementation having the same accuracy of floats, but with halved bit-size. This means that a Posit Processing Unit (PPU) reduces by a factor higher than 2 the data transfer and storage complexity of ML accelerators. We also prove that a LUT-based complete tabulated implementation of a PPU for a 8-bit requires just 64 kB storage size, compliant with memory-constrained devices.

Index Terms - *k*-Nearest Neighbors (*k*-NN), Alternative Real Representation, Posits, Machine Learning (ML) Accelerator

I. INTRODUCTION

Autonomous driving is a safety critical application, as specified also in functional safety standards like ISO26262, with strict requirements in terms of real-time (both throughput and latency) [1, 2]. In Levels 1 and 2 of the SAE autonomous driving scale [1] just an assistance to human driver is needed. Hence, signal processing based on deterministic algorithms is still enough, e.g. FFT-based processing of Frequency Modulated Continuous Wave Radar (FMCW) as done in [1]. Instead, for high autonomous driving levels, from L3 to L5, the complexity of the scenario and the needs of signal processing are very high, not only for sensing, but also for localization, navigation, decision and actuation. As consequence, in recent state-of-art Machine Learning (ML) signal processing is proposed to be used on-board of vehicles [1-4]. ML approaches have reached the state-of-art in several signal processing domains [4-7] like image processing, segmentation, classification and broader computer vision. Tasks such as scene understanding (image segmentation, region-of-interest extraction, sub-scene classification, etc.) must be done on-board the vehicle, since cloud-based computing scenarios (where the processing is done on remote cloud server and on-board there is only a client generating requests to the server) suffers of several issues: privacy, authentication, integrity and connection latency and contention or even communication unavailability in uncovered areas (highway tunnels, etc). On board ML computing can be done only if the computational algorithm complexity is not too high, and a performing HW is adopted. Hence, on-board computing units for ML should be optimized in terms of the ratio between processing throughput performance and resources (memory, bandwidth, power consumption, ...) [7-9]. This is the trend that also big industrial players are following like Google, Nvidia or Intel, that are trying to enter in the autonomous driving market, or the recently announced Full Self Driving (FSD) chip from Tesla. This topic is also the core of the automotive stream in the H2020 European Processor Initiative (embedded HPC for autonomous driving with BMW as main technology end-user [9, 10]) funding this work.

To address the above issues new computing arithmetic styles are appearing in research [11-20] overcoming the classic fixed-point (INT) vs. IEEE-754 floating-point duality in case of embedded DNN (Deep Neural Networks) signal processing. Just as an example, Intel is proposing BFLOAT16 (Brain Floating Point), that has same number of exponent bits of the single-precision floating point allowing in this way to replace binary32 in practical uses although with less precision. BFLOAT16 are supported in Google TensorFlow software, in Google Tensor Processing Units (TPU) and Intel AI processors. Intel is also proposing flexpoint [11, 12] in which exponent information is shared among a group of numbers. NVIDIA for its latest Turing architecture is supporting the concurrent execution of floating point and integer instructions in the Turing SM such as Float32/Float16 and INT32/8/4 precision modes for inferencing workloads that can tolerate quantization [13]. The Tesla FSD chip exploits a neural processing units using 8-bit by 8-bit integer multiply and a 32-bit integer addition. Transprecision computing for DNN is also proposed in state of art by academia [14] and industry, e.g. IBM and Greenwaves in [15]. Signal processing sparsity has been exploited recently [16, 17] to achieve a compression of ML complexity to reach real-time computing on edge devices. However, the deep compression in [16] is paid in terms of accuracy reduction, e.g. 76.6% Top-1 (far from the requirements, typically above 95%, of functional safety applications) on the *Imagenet* object classification challenge. Quantized neural networks are proposed in [18], where using data sets such as MNIST, CIFAR-10, and ImageNet, weights and activations are reduced to 1-bit or 2-bit but the top-1 accuracy is limited to 51%. Recently, a novel way to represent real numbers, called Posit, has been proposed [19, 20]. Basically, the Posit format can be thought as a compressed floating-point representation, where more mantissa bits are used for small numbers, and less mantissa bits for large numbers, within a fixed-length format (the exponent bits adapt accordingly, to maintain the format fixed in length). In this work we present the results obtained when exploiting the Posit format, with a new proposed *cppPosit* library, and exploiting also tabulated look-up table (LUT) based HW calculation, on a widely used ML algorithm, like the *k*-NN classifier [4].

II. The Posit Format

The Posit representation [19, 20] is depicted in Fig. 1. A Posit contains a maximum of four fields: the sign bit, the regime field, the exponent field and the fraction fields. The fields are variable length with priority given to the encoding of regime, then exponent and finally fraction. The maximum length of the exponent is decided a-priori, together with the total length in bits. These two lengths characterize different types of Posit representations. The length of the regime field is determined using a run-length method: the number of consecutive 0 after the sign bit and before the first 1 bit is the regime length (a regime field can be also made by a sequence of 1, until the first 0 is encountered: in that case the number of consecutive 1 is the regime length, but this time its value is negative). Once the length of the regime is known, the length of the mantissa can be determined, as the number of remaining bits (after skipping the exponent bits). The formula that allows retrieving the real number is in [20] and two examples of its application are shown in Fig. 2. Please observe how the two Posits representations shown in the figure have a different number of bits reserved for the fraction field (8 and 9, respectively), having different lengths for the regime fields (4 vs 3).

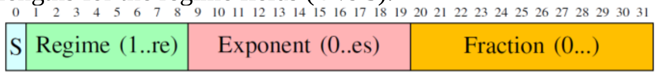

Fig. 1. An example of Posit data type.

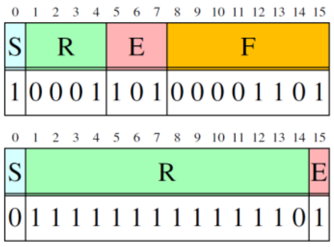

Fig. 2. Two examples of 16-bit Posit with 3 bits for exponent (es=3). In the upper the numerical value is: $-256^{-3} \cdot 2^{5} \cdot (1 + 13/256)$ (13/256 is the value of the fraction, 1 + 13/256 is the value of the mantissa). The final value is therefore $-1.907348 \times 10^{-6} \cdot (1 + 13/256) \cong -2.0042 \times 10^{-6}$. In the lower the numerical value is: $+256^{+12} \cdot 2^{4} \cdot (1 + 0)$ (since the fractional part of the mantissa is missing, we set it to zero). The final value is therefore $2^{16} \cdot 2^{4} \cong 1.2676506 \times 10^{30}$. The second example allows to clarify that: i) the fractional part can be missing, ii) the exponent field can be shorter than its maximum size (in that case the missing bits are assumed zero: the exponent 4 comes from the reconstructed exponent field 100).

Posits can be conveniently put on a circle sharing the concept of projection of reals over a circle, but different design decisions allow to implement Posit operation without imposing the use of a LUT. In Fig. 3 the circle for a 4-bit Posit is presented, in the case of 1 bit for the exponent.

Posits enjoy many really interesting properties, such as:

- Unique representation for zero
- No representations wasted for Not-A-Number (NaN). When using Posits, an exception is raised instead of reserving representations for NaNs. The IEEE 754 standard wastes a lot of representations for NaNs, which makes also the HW for comparing floats complex.
- No need to support unnormalized numbers (which, instead, are generally introduced in floats to augment the accuracy around zero, but makes the FPU implementation significantly more complex).

Even more interestingly, Posits are sorted like signed integers, when the latter are represented using the two's complement. Thus, comparing two Posits can be done in ALU by *type re-casting* to signed integers: negative Posits are expressed using complement two as integers, and the other three fields allow direct ordering. We think the brightest idea of the Posit representation is to *reserve more bits to the mantissa for small real numbers* (close to zero) and *less for large real numbers*, within a fixed length format (the total length is fixed, although the length of the regime and that of the mantissa vary). Posit can also be viewed as a (lossy) compressed version of a float.

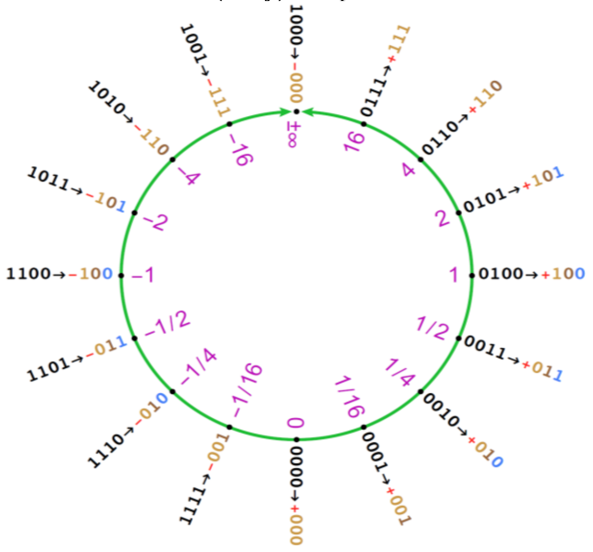

Fig. 3. Posit circle when the total number of bits is 4 and the number of exponent bits is just 1. Observe how the mantissa is almost 1 bit in this case (the last blue bit, when present).

III. The cppPosit Library

In this work we present the implementation of a new C++11 general purpose Open Source library, called *cppPosit* available on github. It is released with a generic programming approach and C++ traits in order to achieve compactness of representation and speed. The library supports Posits having total number of bits ranging from 4 to 64, and supports many variants of the Posits as controlled by the template parameters:

```
template <class T, int totalbits, int
esbits, class FT, PositSpec>
```

where class T is the storage type for the Posit itself (a signed integer data type, such as `int8`); totalbits is the number of bits of the posit. This number can be less than the storage type size for experimenting with different memory layouts; espbits is the maximum number of bits of the exponent; class FT is the storage type for the fractional part of the mantissa during manipulation (an unsigned data type, such as `uint16`). The FT data type is useful when performing the four elementary operations on unpacked Posits – see below –). The presence of NaN is optional.

A. Different operations performed at different levels

The *cppPosit* library uses different Posits represented at different decoding levels.

We defined 4 level of operations for working with Posits:

1. At level 1, operations are performed manipulating directly the bits of the encoding. The cost is the one of integer ALU. Examples of operations that work at level 1 are:
    - 1/x (inversion)
    - –x (unary minus) and |x| (absolute value)
    - x < y, x<=y, x > y, x >= y (comparisons)
    - 1-x (for espbits=0 and x in [0,1])
    - Pseudosigmoid (for esbits=0)
2. At level 2, Posit is unpacked on its underlying fields (sign, regime, exponent, and fraction), without building the complete exponent. The operations are done on such fields and the cost comprises the encoding and decoding. Examples of operations are:
    - x/2 (computing the half of x)
    - 2*x (doubling x)
3. At level 3 we have a fully unpacked version (sign, exponent, fraction). In addition to level 2 operations it is necessary to compute the full exponent. Examples are:
    - convert to/from float, posit or fixed point
4. At level 4 the unpacked version is used to perform the operations in two possible ways:
    - Software floating point
    - Hardware floating point

In the *cppPosit* library everything is template based, this is one of the most important advantages of this library. *cppPosit* defines three key types: Posit (level 1), UnpackedLow (level 2), Unpacked (level 3). Unpacked is parametrized to the type of the fraction (mantissa) and it can handle the conversion to/from any type of IEEE floating point number (expressed via trait) and Posit. A specialized class PostF provides all operations as Level 4 over single or double. In the next section we present an implementation of the *k*-NN which can work with different data types, in particular with Posits and float.

B. Tabulated Posits

When the total number of bits for the considered Posit is lower than or equal to 14, Posits can be tabulated. This allows to speed-up the computation in architecture that still do not have a HW PPU (Posit Processing Unit). To save memory, c*ppPosit* uses some tricks, such as using a single LUT to store both the result of the sum and the difference between two Posits. This is possible due to the fact that the sum is symmetric (so the values are stored on the main diagonal and above it), while the difference is antisymmetric (thus we can store its values below the diagonal). Finally, to avoid the need of the tables for multiplication and division we have tabulated both the natural logarithm and exponential function. When we have to compute the product between Posits, we first compute the logarithms of both, then we sum these values, finally we compute the exponential function of their sum.

A similar trick is used for the division (this time the difference between the two logarithms is computed). Following this strategy, we are able to store everything is needed for the four elementary operations into a single square LUT of size $X*X$, plus two vectors of length X, see Table 1, where X is the total number of bits of the Posit. For Posit8 or 10 the whole computation if tabulated can be saved in cache memory or on-chip SRAM buffer, thus allowing for high-speed computation. For Posit8 the storage size is compliant with memory-constrained units.

Tab. 1 Memory required to store the single LUT as a function of X (total number of bits of the Posit).

Total bits (X)	Storage type bits (b)	Per-table occupation
8	8	64KB
10	16	2MB
12	16	32MB
14	16	512MB
16	16	8GB

IV. Implementing The *k*-NN With Posits Support

The *k*-Nearest-Neighbors (*k*-NN) is a simple learning algorithm used for classification and regression problems. Regarding classification, a new object is classified by a majority vote of its neighbors, in particular the object is assigned to the class most common among its *k* nearest neighbors. The position of each object in *n*-dimension space is determined by its *n*-features. The algorithm can be summarized as:

- A positive integer *k* is specified, along with a new object
- It selects the *k* entries in the dataset which are closest to the new object (e.g. using the euclidean distance)
- It finds the most voted class of these entries. That label will be how the object is classified

The best choice of *k* depends upon the data. Larger values of *k* reduce effect of the noise on the classification, but make boundaries between classes less distinct. A good *k* can be selected by various heuristic techniques.

The *k*-NN can also be used for regression, in this case the output is the property value for the object. This value is the average of the values of its *k* nearest neighbors. We have implemented both a C++ library for Posit support (*cppPosit*) and a generic k-NN algorithm, able to work both with floats and Posits, see Fig. 4. In particular, we extended the nanoflann library (https://github.com/jlblancoc/nanoflann) to operate with Posits. For small-size Posit (up to 12 bits) we have also considered the tabulated version.

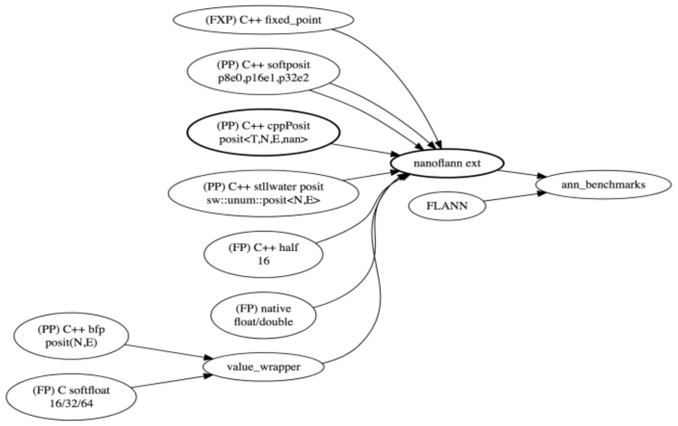
(FXP) C++ fixed_point
(PP) C++ softposit p8e0,p16e1,p32e2
(PP) C++ cppPosit posit<T,N,E,nan>
(PP) C++ stllwater posit sw::unum::posit<N,E>
(FP) C++ half 16
(FP) native float/double
(PP) C++ bfp posit(N,E)
(FP) C softfloat 16/32/64
value_wrapper
nanoflann ext
FLANN
ann_benchmarks

Fig. 4. SW architecture of the implemented *k*-NN library able to run with different data types.

V. Experimental Results

We have tested our implementation on three well-known datasets [4]: sift-128-euclidean; mnist-784-euclidean; fashion-mnist-784-euclidean. Table 2 and Fig. 5 show the performance of different data types, for data set usually used as classification benchmarks in literature (e.g. fashion, mnist and sift), when changing the scaling factor from 0 to 1. The scaling factor parameter rescales the whole dataset. Scaling = 1 means no scaling (thus the original dataset is used in that case). On the other cases, a scaled version of the original dataset is used (thus the variability on the datasets is reduced: this allows data types with lower dynamic to accommodate the observations without truncation). Table 2 shows that Posit16 with 3 bits of exponent works very well. From Fig. 5, the k-NN with a 16-bit Posit with three bits of exponent attains performance close to a float32 and an 8-bit Posit outperforms float16. The achieved results show that Posit16-3 can ensure the same precision (in pattern recognition precision is the fraction of relevant instances among the retrieved instances) than a Float32 on the original data set (Scaling=1). Moreover, Posit8-1 outperforms Float16 since it achieves higher precision for the same scaling factor. Applying an appropriate scaling factor to the dataset values we can adapt the test to types with smaller range. For the *k*-NN implementations the algorithm chosen for the test is nanoflann, that is a pure-template version of the more widespread FLANN [4]. Although nanoflann is parametrized over generic types it required some patching for supporting non-floating point values, with specific care in the accumulation of vector norms. For testing purposes with ann-benchmarks we created a library that contains the nanoflann templates instantiated with each of the relevant types.

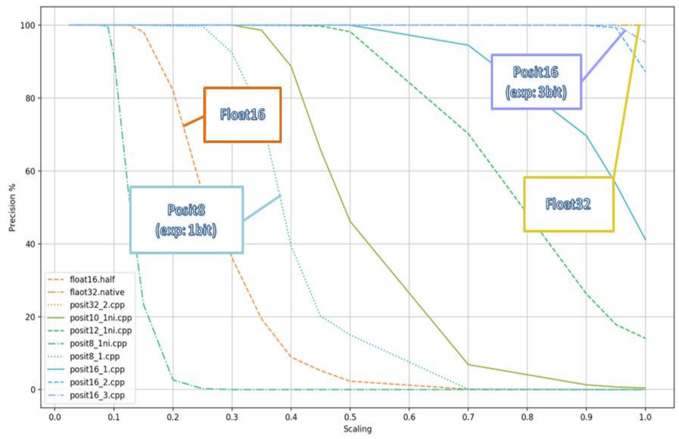

Fig. 5. Precision as a function of the scaling factor, Mnist dataset (similar results for the other benchmark data set)

Tab. 2. Precision obtained on the three datasets, when using float32, Posit16 and Posit32. Scaling factor=1.

Type	Precision (Fashion)	Precision (Mnist)	Precision (Sift)
posit16_2.cpp	87,21%	92,73%	98%
posit16_3.cpp	95,35%	97,73%	99%
posit32_2.cpp	100%	100%	100%
float32_native	100%	100%	100%

VI. Conclusions

This work compared the performance of a *k*-NN classifier, when using Posit and float, showing the benefits introduced by the former. For example, using a PPU instead of floats for *k*-NN classification (on known data set benchmarks), a Posit16 achieved the same accuracy of Floats32 while a Posit8 outperformed Floats16 (that in literature has been proposed as alternative to Floats32 for artificial intelligence applications). This is a remarkable result, not only for saving storage space, but also to better exploit CPU vectorization, all levels of cache, and to increase the bandwidth of data transfer between CPU and RAM to contrast the memory wall phenomenon. As showed in Table 1, a LUT-based tabulated implementation of a PPU for Posit8 requires a 64 kB storage size, compliant with memory-constrained embedded devices. The impact of this work is thus high, since beside autonomous driving there are many safety-critical applications where the accuracy of ML-based decisions is an issue but low-complexity/real-time is needed (robotics, industry4.0, avionics). As future work, we are considering the implementation of fused dot product, the high-level synthesis in HDL starting from the *cppPosit* SW library for HW design of a PPU, and the use of Posits in DNNs.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826647

References

[1] S. Saponara, et al., "Radar-on-chip/in-package in autonomous driving vehicles and intelligent transport systems: opportunities and challenges", *IEEE Signal Processing Magazine*, 36 (5), 2019

[2] L. Lo Bello, et al., "Recent advances and trends in on-board embedded and networked automotive systems", *IEEE Tran. Ind. Inf.*, 15 (2), 2019

[3] "From Signal Processing to Machine Learning", in Digital Signal Processing with Kernel Methods, by J. Royo-Alvarez et al, Wiley, 2018

[4] M. Muja et al.,"Scalable nearest neighbor algorithms for high dimensional data", *IEEE Trans. Pattern Analysis and Mach. Int.*, 36 (11), 2014

[5] T. Bubolz et al., "Quality and Energy-Aware HEVC Transrating Based on Machine Learning", *IEEE Tran. Circ. and Syst. I,* 66 (6), 2019

[6] Li Du et al., "A Reconfigurable 64-Dimension K-Means Clustering Accelerator with Adaptive Overflow Control", *IEEE Trans. Circ. and Sys. II* , 2019, doi 10.1109/TCSII.2019.2922657

[7] P. Nousi, et al., "Convolutional Neural Networks for visual information analysis with limited computing resources" *IEEE ICIP2018*, pp.321-325

[8] Yu Cheng et al., "Model Compression and Acceleration for Deep Neural Networks", *IEEE Signal Proc. Mag.*, pp. 126-136, 35 (1), 2018

[9] D. Reinhardt et al., "High performance processor architecture for automotive large scaled integrated systems within the European Processor Initiative research project", *SAE Tech. Paper 2019-01-0118*

[10] https://www.european-processor-initiative.eu/

[11] U. Köster et al. "Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks", *NIPS 2017*, pp. 1740-1750

[12] V. Popescu et al., "Flexpoint: predictive numerics for deep learning", *IEEE Symposium on Computer Arithmetics, 2018*

[13] "NVIDIA TURING GPU Architecture, graphics reinvented", White paper n. WP-09183-001_v01, pp. 1-80, 2018

[14] G. Tagliavini et al., "FlexFloat: A Software Library for Transprecision Computing", *IEEE Trans. on CAD of Int. Cir. and Syst.* 2019

[15] A. Malossi et al., "The transprecision computing paradigm: concept, design, and applications", *IEEE DATE 2018*, pp. 1105-1110

[16] G. Venkatesh et al., "Accelerating Deep Convolutional Networks Using Low-Precision and Sparsity", *IEEE ICASSP 2017*

[17] G. Srivastava et al., "Joint optimization of quantization and structured sparsity for compressed deep neural networks", *ICASSP 2019*

[18] I. Hubara, et al., "Quantized neural networks: training neural networks with low precision weights and activations", *J. ML Research*, 18 (1), 2017

[19] M. Cococcioni, et al., "Exploiting posit arithmetic for Deep Neural Networks in autonomous driving applications," *IEEE Automotive 2018*

[20] J. L. Gustafson et al., "Beating floating point at its own game: Posit arithmetic," *Supercomp. Frontiers and Innov.*, 4 (2), 2017