Next Generation Arithmetic for Edge Computing

Andre Guntoro*, Cecilia De La Parra*, Farhad Merchant†, Florent De Dinechin‡,
John L. Gustafson§, Martin Langhammer¶, Rainer Leupers†, Sangeeth Nambiar‖
*Corporate Research, Robert Bosch GmbH, Germany
†Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Germany
‡Univ Lyon, INSA Lyon, Inria, CITI
§National University of Singapore, Singapore
¶Intel Corporation
‖Bosch Research and Technology Centre - India, Bangalore
{Andre.Guntoro, Cecilia.DeLaParra}@de.bosch.com, Florent.de-Dinechin@insa-lyon.fr, john.gustafson@nus.edu.sg,
martin.langhammer@intel.com, Sangeeth.Nambiar@in.bosch.com, {farhad.merchant, leupers}@ice.rwth-aachen.de

***Abstract*—Arithmetic is a key component and is ubiquitous in today's digital world, ranging from embedded to high-performance computing systems. With machine learning at the fore in a wide range of application domains from wearables to automotive to avionics to weather prediction, sufficiently accurate yet low-cost arithmetic is the need for the day. Recently, there have been several advances in the domain of computer arithmetic, which includes high-precision anchored numbers from ARM, posit arithmetic, bfloat16, etc. as an alternative to IEEE 754-2008 compliant arithmetic. Optimizations on fixed-point and integer arithmetic are also being pursued actively for low-power computing architectures. Furthermore, approximate computing and transprecision/mixed-precision computing have been exciting areas of research forever. While academic research in the domain of computer arithmetic has a long history, industrial adoption of some of these new data types and techniques is in its early stages and expected to increase in the future. bfloat16 is an excellent example for this. In this paper, we bring academia and industry together to discuss the latest results and future directions for research in the domain of next-generation computer arithmetic, especially for edge computing.**

***Index Terms*—Computer arithmetic, floating point arithmetic, mixed-precision arithmetic, IEEE 754-2008 standard**

I. Introduction

Computer arithmetic is ubiquitous in applications ranging from an embedded domain such as smartphones to high-performance computing (HPC) applications like weather modeling. Specifically, in embedded systems, since the platforms have performance limitations due to limited power and area budgets, using appropriate arithmetic is desirable. The constraints on power and area footprints combined with demand for high performance in applications like edge computing in internet-of-things (IoT) have created tremendous challenges for researchers and computer architects [1]. Over the years in response to this challenge, researchers have developed hardware-efficient implementations of computer arithmetic [2]. Recently, there have been attempts to design new representations to replace the existing ones [3].

Major semiconductor manufacturers involved in efficient domain-specific accelerator designs are investigating new arithmetic formats for their applications of interest [4] [5]. For example, the recently proposed *bfloat16* format supports similar dynamic range as single-precision IEEE 754 compliant floating point number, but with lower precision. The bfloat16 format is supported in several products by Intel, Google, and ARM. Similarly, several academic researchers have designed and developed software and hardware tools that can automatically generate efficient arithmetic hardware. The FloPoCo project is one such example. In this paper, we describe and review the following complementing approaches that lead to an optimum computer arithmetic for edge computing:

- Firstly, we describe an application-specific arithmetic where the focus is on operator transformation and computing exactly required arithmetic.
- Secondly, we describe an arithmetic architecture and approaches for efficiently implementing them on field programmable gate arrays (FPGAs) mainly targeting artificial intelligence (AI) applications.
- Thirdly, we describe a hardware-oriented approximate computing for speech recognition and keyword spotting.
- Finally, we discuss a new datatype called *posit arithmetic* that is proposed as a drop-in replacement for IEEE 754 format and arithmetic.

The rest of the paper is organized as follows. In Section II, we discuss application-specific arithmetic. FPGA implementation and support for arithmetic is discussed in Section III. Applications of approximate computing are described in section IV. Posit arithmetic and its hardware cost are discussed in Section Vas an example of alternate number formats. Finally, we conclude our work in Section VI.

II. Application-specific arithmetic design

Conventional arithmetic units designed for general-purpose processors have to be most generally useful.A good illustration is how the fused multiply-and-add became the floating-point unit of choice at the turn of the century: it could replace an adder and a multiplier, but also enable efficient and flexible implementations of division, square root, elementary functions, etc. [6], while improving the performance on linear algebra computations (the measure of all things in high-performance computing). This particular technical choice is now being reconsidered for energy-efficiency reason, but it remains that

general-purpose processors need general-purpose arithmetic units.

In contrast, there are two edge-computing contexts in which application-specific arithmetic units may be better than standard, general-purpose ones. The first is, of course, the design of application-specific circuits, in particular for the increasingly compute-intensive wireless interfaces, or for machine-learning accelerators. The second is FPGA-based reconfigurable computing, which in some applications will offer the best trade-off between cost, energy efficiency, and performance. Designing operators that match the needs of such application-specific contexts is the subject of this section.

There is, by definition, an infinite number of application-specific operators of interest. Therefore, it makes sense to automate their design as much as possible. This is the primary objective of the FloPoCo framework [1]. This section attempts to review the productive hardware arithmetic paradigm that has emerged during the development of FloPoCo: on one side, the open-ended generation of over-parameterized operators that attempt to compute just right at all levels and on the other side, the use of generic hardware frameworks and optimization methodologies.

A. Opportunities of application-specific arithmetic

A first, obvious approach to tailor the arithmetic to the needs of an application is to tailor the number formats. For example, if 17 bits are enough at some point, then the data format should be 17 bits and not 32. Therefore, all application-specific operators should be parameterized in the precision and range of their inputs and outputs. Further, in this section we review several other approaches for tailoring the arithmetic to the needs of a specific application.

Operator specialization consists in optimizing the structure of an operator such that it can leverage some useful property if its inputs. The most classical example is multiplication by a constant, which has been extensively studied. More subtly, a square requires fewer bit-level operations to compute than a multiplication. An operator can also be specialized for a specific sub-range of its inputs if the application dictates that it will only be used in that sub-range. There are many other such specialization opportunities.

Operator fusion involves considering a compound mathematical expression such as $\frac{x}{\sqrt{x^2+y^2}}$ as a single operator to implement. It goes beyond the specialization of the squares as mentioned in the previous paragraph.For such algebraic expressions, there exists generic techniques that will directly derive a bit-level operator [7].

Function approximation is another generic technique that enables the automatic construction of hardware for a different class of mathematical objects, that is (somehow) continuously derivable functions of one variable up to a certain order. It is a very useful block in some situations of operator fusion or specialization, but also in the design of coarser operators such as elementary functions. FloPoCo includes several generic approximators: by using plain tabulation, by using only tables and additions, or by using multipliers additionally, thanks to polynomial approximation.

Operator sharing is a classical optimization technique. In the arithmetic hardware context, a specific opportunity is to look for intermediate computations that can be used by several subsequent computations. A well-researched example is the multiple constant multiplication problem [8].

Finally, **target-specific optimizations** capture and exploit, in the hardware generator framework, the specific performance capabilities of the targeted hardware technology. For instance, given that the logic of modern FPGAs being based on 6-input look up tables, any technique that exploits pre-computed tables of 64 entries (however random these entries may seem) will be implemented extremely efficiently on FPGAs. This deeply orients algorithmic choices compared to ASIC designs. Another difference between ASIC and FPGA include ripple-carry adders (comparatively faster on the latter with respect to random logic) or the fact that some degrees of freedom (e.g. multiplier size) become constraints due to the reconfigurable arithmetic structure. Section III is discusses the exploration of FPGA-specific optimizations in more detail.

B. Computing just right

One key principles of application-specific design is to systematically correlate accuracy with precision. In a general-purpose processor with a fixed 32-bit data-path, it may make sense to have a 32-bit data that holds information accurate to 17 bits only whereas in an ASIC or FPGA context, it does not. No component should output bits that do not carry useful information. And of course, conversely, no component should be designed to be more accurate than it can express on its output. A nice side-effects of this commonsense rule is that it simplifies the interface: for an operator that could be more or less accurate (be it a multiplier by $\sin \frac{17\pi}{256}$, or an approximator to $\log x$ or a digital filter) there is no need to specify the accuracy, as it should be deduced from the output format. Indeed, for several FloPoCo papers, a central contribution has been a better definition of the core problem thanks to such interface simplifications.

C. A generic methodology for the design of complex operators

All the previous approaches exposes many parameters in the interface of an operator (I/O formats, functional parameters such as the constant in a constant multiplier, which variant of which algorithm to use, etc).

Assembling several components into a larger one, as illustrated by Fig. 1, also involves new parameters such as all the internal precisions required at various steps of the algorithm. Some of these parameters may control a cost/performance trade-off. For instance, the size of the sub-word A in Fig. 1 controls a trade-off between table size and multiplier size. There is nothing wrong in over-parameterizing an architecture generator, it can only make it future-proof. Defining an over-parameterized architecture is the first step of the design of an application-specific operator generator.

[1] http://flopoco.gforge.inria.fr/

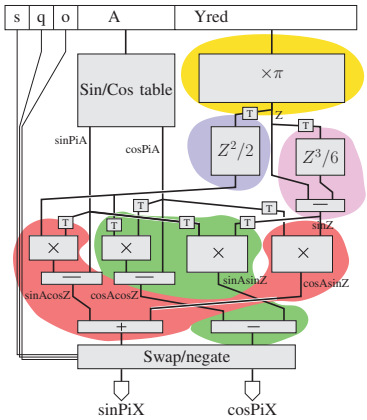

Fig. 1. A parametric architecture for fixed-point sine+cosine [9]. The T boxes describe truncations, the colored bubbles cover operations that are merged in a bit heap. Each bit-width on this figure is computed by the generator, and very few signals have the same bit width.

The second step is, of course, to choose the value of all these parameters. There is no push-button approach here, but three main sub-problems must be solved jointly.

First, we need to be able to compute the accuracy of the architecture as a function of the parameter values through error analysis . Indeed, as we just saw, the output precision entails a constraint on this accuracy. A range of techniques exist and can be mixed and matched, from approximation theory down to a brute force enumeration of all the values of a table, as long as the resulting error analysis can be programmed (with reasonable run-time).

Second, we need to express the cost of the architecture. This will obviously depend on the technology on the target. Here also, inelegant enumerations are just as good as nice closed-form formulae, as long as their run-time remains acceptable.

Finally, we must write a parameter-space exploration that respects the constraints while minimizing the cost.It will define all the internal parameters, and in particular attempt to minimize the intermediate bit-widths, hence computing just right.This exploration can be an ad-hoc program, or use generic optimization techniques such as reductions to the Shortest Vector Problem, Integer Linear Programming, or constraint programming.

An interested reader can find detailed examples of this methodology in many articles describing FloPoCo operators, for instance sine/cosine [9] of IIR filters [1]. We conclude this section with an overview of another high-level optimization framework specifically designed for application-specific arithmetic.

D. The Bit Heap generic arithmetic framework

A bit heap is an arbitrary sum of weighted bits, a generalization of the bit arrays classically used in multiplier design.In

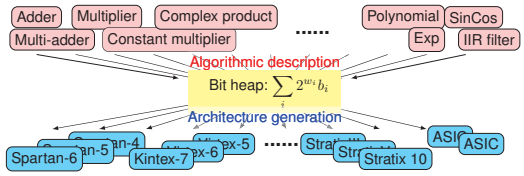

Fig. 2. A bit-heap-centric view on operator generation

FloPoCo, they have been used since 2013 [10] to capture summations in a surprisingly large number of operators, including sums of products and more generally polynomials of several variables, but also all sorts of sums of tabulated values, for instance in table-based FIR and IIR filters [1] or in multipartite table methods for function evaluation [11]. They is an elegant way of decoupling (Fig. 2) the description of an arithmetic computation (as a sum of weighted bits) from the generation of target-optimized hardware that computes this sum. It is not uncommon for a complex operator such as an elementary function to involve several bit heaps (see Fig. 1). Having an open-source bit-heap framework in FloPoCo has also renewed interest in bit array compression, improving the state of the art with an ILP-based heuristic [12] that, in return, improves the performance of a large number of FloPoCo operators.

III. FPGA Based Arithmetic

The most recent FPGA architectures have introduced new levels of embedded floating point (FP) performance, with tens of TFLOPs now available across a wide range of device sizes. The emergence of AI/Machine Learning as the highest profile FPGA application has changed the focus from signal processing and embedded calculations supported by FP32 to smaller FP precisions, such as bfloat16 for training and FP16 for inference. As an example, each Intel Agilex DSP Block [13] contains a FP32 multiplier-adder pair that can be decomposed into two smaller precision pairs; FP16, bfloat16, and a third FP19 {1,8,10} format which can be used for both training and inference. One member of the new Agilex device family contains almost 9000 DSPs; at a clock rate of 750MHz this provides up to 25TFLOPs performance.

Ristretto [14] showed that smaller precisions, especially combined with a flexible block floating point method (described as dynamic fixed-point), could provide effective AI performance with significantly reduced precisions. This research is well suited for FPGA implementations, where an almost unlimited level of flexibility is available to implement small precision soft multipliers.

In the Edge, where even lower precision arithmetic is required for inference, new FPGA EDA flows can implement 100 TFLOPs+ of soft logic-based compute power. These include new synthesis, clustering, and packing methodologies – collectively known as Fractal Synthesis [15] - that allow a near 100% logic use of the FPGA for arithmetic, while maintaining the clock rates of a small example design. Improved soft multiplier mapping algorithms [16] can also balance logic and

routing resources better, and the results can still be packed further by Fractal Synthesis. All of the soft logic methods can be used simultaneously with the embedded FP operators in the DSP Blocks, making the FPGA the most flexible, and amongst the highest performing AI platform available.

One significant limitation of the FPGA is high density fitting, especially with a performant timing closure. While a design consisting of random logic can top 80% logic utilization, soft arithmetic is more typically 60%-70% full. Coupled with very high compute densities required, the known methods of mapping multipliers to soft logic give an ineffective result in terms of PPA (power, performance, area). Two new algorithms were developed to address this: multiplier regularization, which balances the soft logic and soft routing for efficient multiplier implementation, and Fractal Synthesis, which packs carry chains and the logic associated with them into the FPGA with a deterministic result. Although a detailed description of these two methods is beyond the scope of this paper, we will give an overview here, including a simple example in the case of multiplier regularization.

An FPGA can be considered as a relatively fine grained array, but for low precision arithmetic, can appear to be very coarse grained. We will use the example of a 3x3 multiplier to illustrate our approach. Using the pencil and paper method, two inputs a_2, a_1, a_0 and b_2, b_1, b_0 produce 3 partial products of 3 bits each, the second and third of which are arithmetically shifted one bit to the left of the first and second, respectively. This is illustrated by Fig. 3. The term $p_{1,0}$ is the bit 0 of the multiplicand (the least significant bit) ANDed with bit 1 of the multiplier.

The issues can be seen by inspection. FPGAs contain two input ripple carry adders, yet this arrangement leads to three inputs after the second column. The number of independent inputs per column is grossly unbalanced, varying from two to six bits.

We can, however, extract components from the deepest column and calculate them in out of band functions, then refactoring these across the entire chain to the left, creating a two input chain. First, we restate two entries of column 2 with the function $(p_{0,2} \oplus p_{1,1})$, which is the redundant sum compression of those input bits. The four input bits (a_2,a_1,b_1, and b_0) can be used in the same adaptive logic module (ALM) used to calculate the redundant carry function $(a_2 \cdot b_0 \cdot a_1 \cdot b_1)$. Although moving the redundant carry function into column 3 creates another column of three bits, the redundant carry can be refactored to use the redundant sum calculation for column 3, or $(a_2 \cdot b_0 \cdot a_1 \cdot b_1) \oplus (a_2 \cdot b_1)$.

There is, however, another condition that needs to be considered. If $a_2 \cdot b_0$, $a_1 \cdot b_1$, and $a_2 \cdot b_1$ are all '1', then a carry will be produced in column 4. Restated, if the redundant sum in column 3 is $(a_2 \cdot b_0 \cdot a_1 \cdot b_1) \oplus (a_2 \cdot b_1)$, the following redundant carry will be $(a_2 \cdot b_0 \cdot a_1 \cdot b_1) \cdot (a_2 \cdot b_1)$. The redundant sum can be XORed with $a_2 \cdot b_1$ in the cell of column 4, generating $(a_2 \cdot b_0 \cdot a_1 \cdot b_1) \oplus (a_2 \cdot b_1) \oplus (a_2 \cdot b_1)$, or $(a_2 \cdot b_0 \cdot a_1 \cdot b_1)$. The redundant carry is then $(a_2 \cdot b_0 \cdot a_1 \cdot b_1 \cdot a_2 \cdot b_1)$, which can be reduced to $(a_2 \cdot b_0 \cdot a_1 \cdot b_1)$. Note that this is identical to the previous redundant sum.

The unsigned 3x3 multiplier can now be mapped to a single 3 ALM carry chain, with a single out of band ALM. The routing and logic are now balanced, with 6 independent inputs over the 4 ALMs. See Fig. 4 for details. This method can be expanded to larger multipliers, as well as the summation of multiple carry vectors such as those found in a soft multiplier or dot products [16].

Now that we have optimized the mapping of small multipliers to the FPGA soft logic resources, we must address the fitting challenges of the many independent short carry chains that these functions contain. Fitting is a variant of the well known bin packing problem. The physical carry chains on the FPGA will be organized in fixed granularities, and we need to find a way to pack the many more logical carry chain segments efficiently. The low fitting rates that are generally seen for these soft arithmetic arrays underscore that there is rarely a good solution available for this, which is made more challenging as the segments need to be arithmetically separated from each other (typically by the insertion of non-functions), which further reduces the efficiency of the final fit.

We now add a re-synthesis step to the clustering and packing stage, and approach the problem as a combined logic and carry chain fit. If a carry segment cannot fit in the space available, we start to decompose it, followed by other segments that may already exist in a trial fit to that physical chain. Sub-segments that are split off are later placed in remaining gaps after all segments have been placed. Finally, a hard depopulation is used to complete gaps in the physical chain – so that the back end of the tool does not try to alter the arrangement of sub-segments.

This process is iterated exhaustively, rather than simulated annealing, with a seed function to initialize each iteration. Unlike many EDA algorithms, we do not need to keep a record of each solution, only a list of seeds and their final metrics are tracked. The best solution can be quickly re-created using the chosen seed; this reduces RAM and disk usage, and in turn provides a marked improvement in run time.

Access to embedded DSP Blocks are not affected by these soft logic methods, and can still be independently used, for more system flexibility. Alternately, small multipliers can be extracted from the embedded multipliers (this is especially efficient if one of the input operands is shared, as is often the case for deep learning implementations). The dot products can then be realized using soft logic adders packed by Fractal Synthesis. This approach is validated by the Brainwave [17] design, where 92% logic utilization was achieved. This architecture has two components: control comprises 20% of the design at a packing rate of about 80%, and the datapath, which contains 80% of the design with 97% packing.

IV. Approximate Computing Paradigm for Complex Neural Networks

Approximate computing at software level (e.g. quantization and pruning techniques [18]–[20]) and hardware level (e.g.

Column	5	4	3	2	1	0
PP0	0	0	0	$p_{0,2}$	$p_{0,1}$	$p_{0,0}$
PP1	0	0	$p_{1,2}$	$p_{1,1}$	$p_{1,0}$	0
PP2	0	$p_{2,2}$	$p_{2,1}$	$p_{2,0}$	0	0

Fig. 3. 3×3 Multiplier $\{a_2, a_1, a_0\} \times \{b_2, b_1, b_0\}$

Column	5	4	3	2	1	0
PP0	0	$p_{2,2}$	$p_{2,1}$	$p_{2,0}$	$p_{0,1}$	$p_{0,0}$
PP1	0	$AUX_2 \oplus p_{1,2}$	AUX_2	AUX_1	$p_{1,0}$	0

Fig. 4. 3×3 Multiplier in Two Levels with Auxiliary Functions

partial [21]–[23] and full [24], [25] approximations) has been thoroughly investigated in recent years. Energy saving in partial approximation, which typically does not require any retraining, is smaller while not all computation elements are approximated. Full DNN approximation on the other side allows for larger energy saving that can only be achieved through effective retraining [24], [26]. Recent works have shown that approximate computing is a viable approach for more complex tasks [27]. However, few have investigated the effects of approximate computing for other tasks such as speech recognition or keyword spotting, which are typical applications on edge devices.

A. Approximating Multiplications on DNNs

We define the approximate multiplication $\tilde{g}(\cdot)$ between two values a, b as $\tilde{g}(a,b) = a \times b + \epsilon_{a,b}$, where $\epsilon_{a,b}$ is the approximation error, deterministic by nature, and dependent from values a and b. Approximate DNN layers can be mathematically expresed as $\tilde{\mathbf{y}} = \Psi(f(\mathbf{x},\mathbf{w}) + \tilde{g}(\mathbf{x},\mathbf{w}) + \mathbf{b}) = \Psi(\tilde{f}(\mathbf{x},\mathbf{w}) + \mathbf{b})$, where $f(\mathbf{x},\mathbf{w})$ is either a convolutional operation or a matrix multiplication between inputs $\mathbf{x}$ and weights $\mathbf{w}$ in case of a fully connected layer, $\tilde{f}(\mathbf{x},\mathbf{w})$ is the approximate counterpart, $\mathbf{b}$ is the bias and Ψ is a non-linear function. The result of $\tilde{f}(\mathbf{x},\mathbf{w})$ is hereby obtained by introducing the behavioral simulation of a given approximate multiplier in the computation of $f(\mathbf{x},\mathbf{w})$. To achieve this, we implement *ProxSim* [27], a GPU-accelerated simulation framework specialized for approximate computation of convolutional and fully connected layers. We quantize weights, bias, and activations to 8 bits using linear quantization. For our case study, we consider DNNs shown in Table I and 10 randomly selected approximate multipliers from EvoApprox [28] shown in Table II. ResNet20 is used for image classification, whilst KWS-CNN1 and KWS-CNN2 for keyword spotting. MRE and MAE correspond to mean relative error and mean absolute error respectively.

TABLE I
DNN CHARACTERISTICS

DNN	Dataset	Params	MACs	Float	8-bit
ResNet20	CIFAR	274,442	40.8M	91.04	90.34
KWS-CNN1	SCD	69,982	2.5M	91.99	91.90
KWS-CNN2	SCD	179,404	8.6M	92.71	92.60

TABLE II
APPROXIMATE MULTIPLIERS

Multiplier	MRE [%]	MAE	Energy Saving [%]
320	0,03	0,2	0,02
114	1,26	11,2	7,59
302	2,38	22,9	15,49
231	4,94	46,6	22,10
62	6,04	73,7	30,85
163	11,88	165,8	51,90
435	14,34	217,3	56,87
24	16,24	343,4	62,00
195	17,67	283,8	63,08
280	19,45	343,9	68,08

B. DNN Approximation Technique

Accuracy degradation tolerance is generally defined for approximate DNNs. Following [21], [27] we propose a tolerance of 1% and 5% with respect to quantized (8-bit) accuracy for image classification and keyword spotting respectively. We use the cross-entropy loss as cost function for the initial DNN training, i.e.

$$C_\theta(\tilde{Y}) = -\sum_{k=1}^{n} \Gamma_k log \tilde{Y}_k \tag{1}$$

Then, the gradients are updated with respect to the weights through back-propagation, i.e.

$$\mathbf{w}_t = \mathbf{w}_{t-1} + \Delta\mathbf{w} = \mathbf{w}_{t-1} + \nabla \frac{\partial C(\tilde{Y})}{\partial \mathbf{w}_{t-1}} \tag{2}$$

with $\frac{\partial C(\tilde{Y})}{\partial \mathbf{w}} = \frac{\partial C(\tilde{Y})}{\partial \tilde{Y}} \frac{\partial Y}{\partial \mathbf{w}}$. Note that we compute the gradient of Y (with respect to $\mathbf{w}$) instead of $\tilde{Y}$. This is necessary as the gradient of approximate function is undefined and thus we need to estimate it using the accurate counterpart.

C. Evaluation

We evaluate our proposed approximate retraining in *ProxSim*. The detailed results are presented as follow.

1) Task accuracy: We retrain the DNNs over 5 epochs with 10 different multipliers from Table II. The results is presented in Fig. 5. For ResNet20, we observe a uniform accuracy recovery, reaching the defined accuracy tolerance in 70% of the cases, which corresponds to multiplier with energy saving of up to 56,8%. In case of keyword spotting, the accuracy tolerance is reached in all cases, with maximum energy saving of 68%, although the accuracy slightly decreases for approximate multipliers with MRE higher than 2,4%.

2) Effects of data augmentation: Approximate computational units introduce unwanted noise in DNN operations through approximation errors. Such noise, in controlled amounts, acts as a regularizer when training approximate DNNs [29]. Thus, we propose DNN retraining without data augmentation, as this is also another form of regularization through input alteration. By performing data augmentation, the DNN approximation error is then harder to compensate. We compare the results of training with and without data augmentation in Fig. 5. For image classification, we randomly

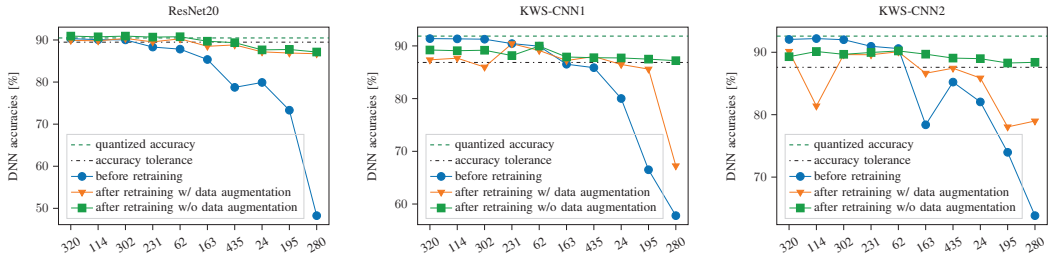

Fig. 5. Task accuracy with 10 different approximate multipliers on 3 DNNs

flip the training samples, and for keyword spotting, we add background noise with a volume of 10% to the initial time series. As observed, data augmentation worsens the accuracy degradation in approximate DNNs, specially for speech recognition tasks.

3) Conclusions: The paradigm of approximate computing delivers promising results for optimizing energy consumption of perception tasks. In this work, we present an unprecedented analysis of hardware-oriented approximate computing for speech recognition tasks, and highlight the role of data augmentation and regularization through approximation error through experiments in DNNs for image recognition and keyword spotting.

V. Fair Hardware Comparison of Posits vs IEEE Floats

Before comparing IEEE floating point format and the emerging *posit* format [3], consider a similar transition that occurred half a century ago, regarding the storage of signed integers on digital computers.

Early binary computers represented signed integers with *sign-magnitude* arithmetic, mimicking the notation people use to write numbers: One bit for the sign, and the remaining bits for the magnitude, written with positional notation $d_n d_{n-1} ... d_0$ where each digit d_i is implicitly multiplied by b^i for a number base b (10 for decimal, 2 for binary). (The ENIAC (1946) even went so far in the direction of human-oriented format that the on-off logic of vacuum tubes was used to represent decimal digits $0-9$ throughout the machine instead of recognizing the much greater efficiency of using 2 as the base of the positional notation.) For multiplication, the sign of the product is the XOR of the signs of the inputs, and the magnitude is simply the integer product of the input magnitudes, performed by early machines using repeated integer add-shift operations.

Addition and subtraction with sign-magnitude representation is less simple. Hardware must accomplish at least the following to add integers i and j to produce a sum k, ignoring the handling of overflow:

if sign(i) = sign(j) **then**
  magnitude(k) = magnitude(i)+magnitude(k)
  sign(k) =sign(i)
else
  if magnitude(i) > magnitude(j) **then**
    magnitude(k) = magnitude(i)−magnitude(k)
    sign(k) =sign(i)
  else
    magnitude(k) = magnitude(j)−magnitude(i)
    sign(k) =sign(j)
  end if
end if

Complications also exist from the redundant representation of 0 as "positive zero" and "negative zero." The comparison operators must make an exception when comparing with zero because it is not a simple matter of checking that all bits are identical. A choice must be made whether the sign of zero is "don't care" or subject to special rules.

A more mathematical approach, *2's complement format*, soon replaced sign-magnitude format because it reduces the above algorithm to a single line: $k = i + j$ where i and j are treated as unsigned integers. Multiplication is as simple as before. Redundant representation of 0 is eliminated, and comparison becomes trivial. Its only downside is that it is *less human-readable*. Someone practiced at reading binary might recognize `00000101` as the number 5 and in sign-magnitude, `10000101` as the number -5, but it is not easy for humans to read the 2's complement for -5: `11111011`. Designers initially used human-oriented parcels of bits or "bit fields" and eventually realized it is more important for the number format to be logic-oriented and hardware-efficient. We can imagine a designer familiar only with sign-magnitude being introduced to 2's complement, and finding it less efficient because of the (erroneously perceived) need to first extract input sign bits and take absolute values to find the magnitudes, then re-encode the result back as 2's complement.

The IEEE 754 Standard for floating-point representation [30] reverted to sign-magnitude format and human-oriented bit parcels for the scale factor and the significand. Posit arithmetic is based on 2's complement principles. Some published comparisons (e.g. [2], [31]) make the mistake just described of re-casting everything into familiar bit parcels and then having to re-encode the result.

We can diagram 2's complement integers on a ring, where the bottom of the ring represents `00...00` for zero, increases

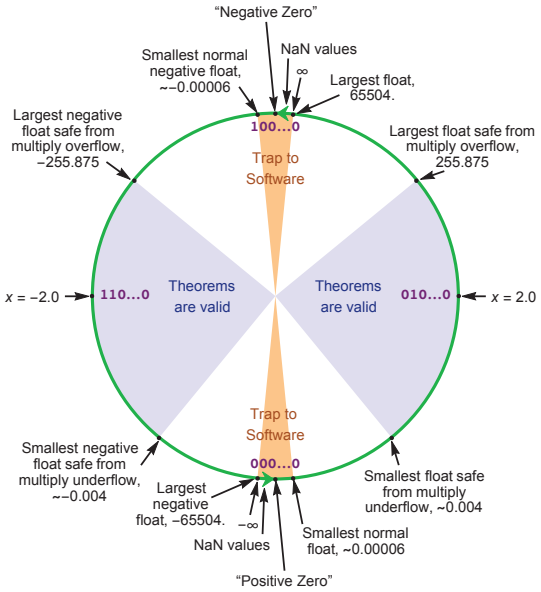

Fig. 6. Ring Plot of 16-Bit Floats

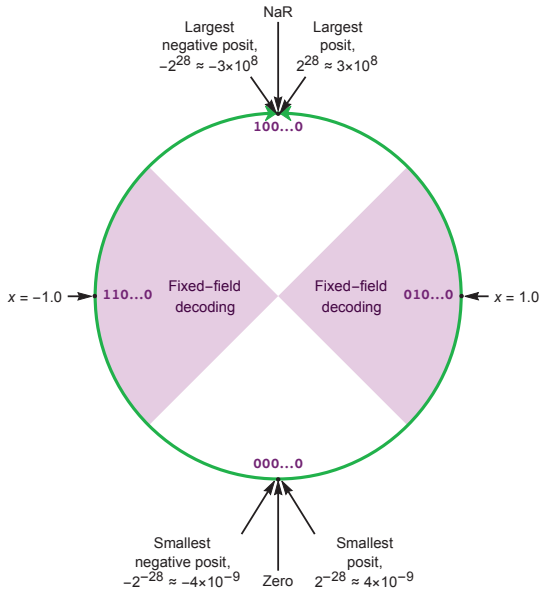

Fig. 7. Ring Plot of 16-Bit Posits

counterclockwise to the largest positive integer `011...11`, which then increments to the most negative integer represented by `100...00` at the top of the ring. It continues a monotonically climbing representation to -1 as `11...11` in the clockwise direction, wrapping to zero. Fig. 6 shows the integer bit strings on the inside of the ring and how the IEEE 16-bit floats map to those bit strings, drawn to scale. A similar diagram holds for all IEEE float sizes. Almost universally, processors do not use hardware for the "Trap to Software" sections where the exponent field is all `0` bits or all `1` bits. The result is that calculations run orders of magnitude slower for about 6 percent of the possible values, which also allows side-channel security attacks because of the effect on system timing [32]. Outside those regions, it is possible to decode floats as "normal" with a sign, exponent, and fraction field. On a SIMD or vector architecture, it is common to use flags to turn off the IEEE rules for the NaN and subnormal regions so that one exception does not cause a massive slowdown in processing the entire list [33].

A theorem taught in numerical analysis courses is one that bounds the relative error caused by computing the product of two floats, $a \times b$. However, it is rarely taught that this theorem and others like it apply to *less than half* the range of possible inputs a and b, shown in the arc regions marked "Theorems are valid." Outside this range, underflow and overflow are possible. For 16-bit floats, the dynamic range is already so limited (about 6×10^{-5} to 7×10^{4}) that they are difficult to use for AI and signal processing; the effective dynamic range is much smaller if we expect to do any multiplies or divides, from $1/256$ to a little less than 256. The restricted dynamic range has led Google to introduce its 16-bit "bfloat," a 32-bit float with the 16 least-significant fraction bits rounded off.

In Fig. 6 we can also note that floats increase monotonically on the right half of the ring but reverse direction for the negative values, the left half. The IEEE 754 Standard requires 22 different kinds of comparison operations because of the NaN exceptions; NaN compares as "not equal" to itself and "unordered" to everything else; Negative zero and positive zero compare as equal. Substantial circuit logic is needed for the comparison of two floats.

In contrast, consider the posit mapping to the ring plot of integers shown in Fig. 7. With only two exception values, there is no need to trap to software; both exceptions have all `0` bits after the first bit. The OR tree takes no more than six logic levels (less than a clock cycle) even for 64-bit posits, and that test can execute in parallel with processing that assumes a non-exception value. Execution times can thus be made data-independent and quick, eliminating both the security hole and the need to disable exceptions for parallel processing. The single Not-a-Real (NaR) value takes care of all non-real outputs, relying on computer languages (and debuggers) to catch such errors during program development.

Reciprocation is symmetric for posits, and negation with 2's complement also works without exception. There is no need for a posit comparison unit separate from the one used for integers; NaR is treated as equal to itself and less than all other numbers.

The shaded arcs in Fig. 7 are regions that can be decoded

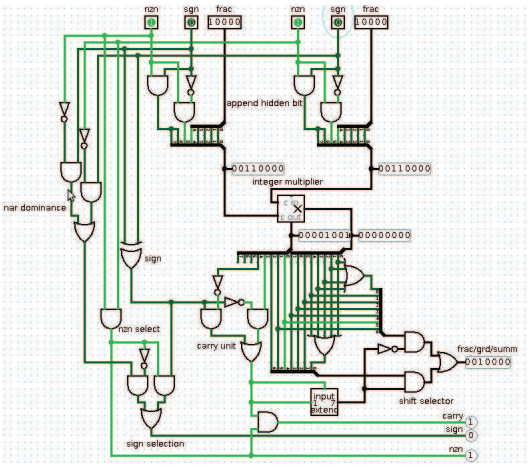

Fig. 8. 8-Bit Posit Multiplier (Yonemoto)

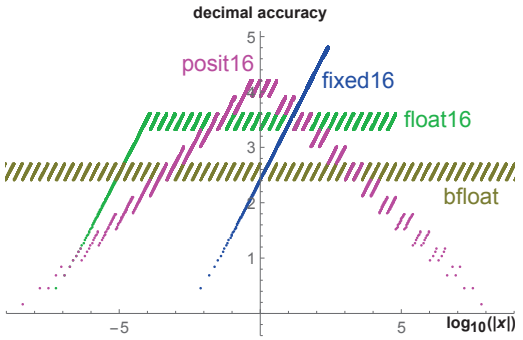

Fig. 9. Accuracy as a function of bit string for 16-bit formats

as easily as floats, because there are exactly two regime bits and a count-leading-zero-or-one operation is not needed. All the bit fields are in a fixed location, but are best decoded with 2's complement. The circuit by Isaac Yonemoto in Fig. 8 shows a multiplier for 8-bit posits With floats, the "hidden bit" is the OR of the exponent bits: 0 for subnormal floats (and zero), and 1 for normal floats. While posits can make use of a similar trick to set the hidden bit to 0 for zero and achieve correct results in the arithmetic logic, Yonemoto's insight was that *the hidden bit means* -2 *for negative posits*. The need for separate circuitry for negative values is eliminated. We can think of the fraction bits added to 01 for positive values, meaning 1, and 10 for negative values, meaning -2, so the significand counts from 1 to 2 for positive values but from -2 to -1 for negative values. The community is still in the early stages of discovering such circuit shortcuts, so we will initially see designers trying to implement posits by first forcing them to look more like floats, then converting back.

A similar economy exists for addition and subtraction. A 16-bit posit has a dynamic range from 2^{-28} to 2^{28} and can thus be converted to a signed fixed-point representation with 58 bits (an extra bit is needed to indicate the sign). The add or subtract logic simply needs to perform an arithmetic shift on the fraction that preserves the sign, add or subtract as integers, and convert the result back to posit form. Up to some precision, single-cycle posit addition and subtraction appear possible. For larger precision posits, it is more economical to align binary points as with floats; 2's complement still avoids some of the conditional testing needed for float addition and subtraction.

Hardware engineers and programmers faced with choosing between fixed-point and floating-point representations now have a third choice: posit representation. Fig. 9 summarizes the decimal accuracy of 16-bit representations as a function of the magnitude (log base 10) of the absolute value of the value represented by the format. Fixed-point (integer) format is the simplest and fastest format, but has very unbalanced accuracy about low magnitudes and a very restricted dynamic range. The float values have flat accuracy except for the subnormal number range on the left, where accuracy tapers to zero. For the most common values in the range of about 0.01 to 100, posits have higher accuracy than IEEE floats and bfloats, but less accuracy outside this dynamic range. For all precisions, float accuracy forms a trapezoidal shape; fixed-point accuracy looks like a triangular ramp upward; and posit accuracy is an isosceles triangle centered at magnitude zero. Depending on the applications, posits often maximize information-per-bit in the Shannon sense, compared to the other formats.

Another way to view the accuracy is as a function of the *bit string* that expresses the value, treated as an integer. For 16-bit representations, the bits range from 0 to 32767 for positive values, as shown in Fig. 10. This figure shows how 16-bit posits have nearly the accuracy of fixed-point representation, but also provide a large dynamic range (almost 17 orders of magnitude, compared to only 9 orders of magnitude for IEEE 754 Standard 16-bit floats in the normal range, and less than 5 orders of magnitude for fixed-point format). The bfloats have a huge dynamic range (about 76 orders of magnitude) but at the cost of less than three decimals of accuracy that forces users to deal with quantization errors.

In summary, comparisons of posit and float hardware complexity need to be careful to note whether the float hardware actually supports IEEE 754 or if the compliance is limited to normal floats only. Posit hardware is slightly more expensive than normals-only float hardware, but substantially simpler and faster than hardware that fully supports all aspects of the IEEE 754 Standard.

VI. Conclusion

In this paper, we presented four complementing approach to build optimum computer arithmetic for edge computing applications. At first, we discussed several approaches in operator transformation for edge computing. The transformations presented in this paper are part of FloPoCo project. Later, field-

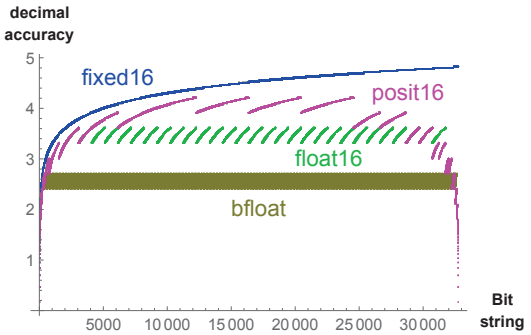

Fig. 10. Accuracy as a function of bit string for 16-bit formats

programmable gate array-based arithmetic frameworks were discussed in detail where the arithmetic support for the hardware platform was discussed. We also discussed approximate computing and applications of the same for complex neural network architectures. It was observed that the paradigm of approximate computing yields significant energy benefits for edge computing devices. Finally, a new data representation called *posit* and its properties were described. Mainly, the hardware costs of posit arithmetic compared to the IEEE 754 compliant arithmetic were evaluated. In future, we continue to experiment and research further in the domain of computer arithmetic to improve their energy and area requirements, especially keeping in mind the requirements of edge computing devices.

References

[1] A. Volkova, M. Istoan, F. de Dinechin, and T. Hilaire, "Towards hardware IIR filters computing just right: Direct form I case study," *IEEE Transactions on Computers*, vol. 68, no. 4, Apr. 2019.

[2] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, "Parameterized posit arithmetic hardware generator," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct 2018, pp. 334–341.

[3] Gustafson and Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov.: Int. J.*, vol. 4, no. 2, pp. 71–86, Jun. 2017. [Online]. Available: https://doi.org/10.14529/jsfi170206

[4] R. Ma, J. Hsu, T. Tan, E. Nurvitadhi, D. Sheffield, R. Pelt, M. Langhammer, J. Sim, A. Dasu, and D. Chiou, "Specializing FGPU for persistent deep learning," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2019, pp. 326–333.

[5] S. Vogel, M. Liang, A. Guntoro, W. Stechele, and G. Ascheid, "Efficient hardware acceleration of CNNs using logarithmic data representation with arbitrary log-base," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18. New York, NY, USA: ACM, 2018, pp. 9:1–9:8. [Online]. Available: http://doi.acm.org/10.1145/3240765.3240803

[6] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*, ser. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[7] M. Ercegovac, "A general hardware-oriented method for evaluation of functions and computations in a digital computer," *IEEE Transactions on Computers*, vol. C-26, no. 7, pp. 667–680, 1977.

[8] M. Kumm, "Optimal constant multiplication using integer linear programming," *Transactions on Circuits and Systems II*, vol. 65, no. 5, 2018.

[9] F. de Dinechin, M. Istoan, and G. Sergent, "Fixed-point trigonometric functions on FPGAs," *SIGARCH Computer Architecture News*, vol. 41, no. 5, pp. 83–88, 2013.

[10] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic core generation using bit heaps," in *Field-Programmable Logic and Applications*, Sep. 2013.

[11] S.-F. Hsiao, P.-H. Wu, C.-S. Wen, and P. K. Meher, "Table size reduction methods for faithfully rounded lookup-table-based multiplierless function evaluation," *Transactions on Circuits and Systems II*, vol. 62, no. 5, pp. 466–470, 2015.

[12] M. Kumm and J. Kappauf, "Advanced compressor tree synthesis for FPGAs," *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1078–1091, 2018.

[13] "Intel® agilex™ FPGA Advanced information Brief." Intel, 2019.

[14] P. Gysel, "Ristretto: Hardware-oriented approximation of convolutional neural networks," 2016.

[15] M. Langhammer, G. Baeckler, and S. Gribok, "Fractal synthesis: Invited tutorial," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: ACM, 2019, pp. 202–211. [Online]. Available: http://doi.acm.org/10.1145/3289602.3293927

[16] M. Langhammer and G. Baeckler, "High density and performance multiplication for FPGA," in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, June 2018, pp. 5–12.

[17] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale DNN processor for real-time AI," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 1–14.

[18] S. Vogel, J. Springer, A. Guntoro, and G. Ascheid, "Self-supervised quantization of pre-trained neural networks for multiplierless acceleration," in *DATE 2019*, 2019.

[19] S. R. Jain, A. Gural, M. Wu, and C. Dick, "Trained uniform quantization for accurate and efficient neural network inference on fixed-point hardware," in *arXiv:1903.08066*, 2019.

[20] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *ICCV 2017*, 2017.

[21] S. Venkataramani et al., "AxNN: Energy-efficient neuromorphic systems using approximate computing," in *ISLPED 2014*, 2014.

[22] Q. Zhang et al., "ApproxANN: An approximate computing framework for artificial neural network," in *DATE 2015*, 2015.

[23] V. Mrazek, Z. Vasıcek, L. Sekanina, M. A. Hanif, and M. Shafique, "ALWANN: automatic layer-wise approximation of deep neural network accelerators without retraining," in *ICCAD 2019*, 2019.

[24] X. He et al., "AxTrain: Hardware-oriented neural network training for approximate inference," in *ISLPED 2018*, 2018.

[25] S. S. Sarwar, S. Venkataramani, A. Ankit, A. Raghunathan, and K. Roy, "Energy-efficient neural computing with approximate multipliers," in *Journal on Emerging Technologies in Computing Systems*, 2018.

[26] Y. Fan, X. Wu, J. Dong, and Z. Qi, "AxDNN: Towards the cross-layer design of approximate DNNs," in *ASPDAC 2019*, 2019.

[27] C. D. la Parra, A. Guntoro, and A. Kumar, "ProxSim: GPU-based simulation framework for cross-layer approximate DNN optimization," in *to appear in DATE 2020*, 2020.

[28] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "EvoApprox8B: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *DATE 2017*, 2017.

[29] C. M. Bishop, "Training with noise is equivalent to tikhonov regularization," *Neural Computing*, 1995.

[30] *IEEE Standard for Floating-Point Arithmetic*, 2008.

[31] Y. Uguen, L. Forget, and F. de Dinechin, "Evaluating the hardware cost of the posit number system," in *29th International Conference on Field-Programmable Logic and Applications (FPL)*, Barcelona, Spain, Sep. 2019. [Online]. Available: https://hal.inria.fr/hal-02130912

[32] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *2015 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2015, pp. 623–639. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2015.44

[33] I. Dooley, L. V. Kalé, and N. Goodwin, "Quantifying the interference caused by subnormal floating-point values," 2006.