NnCore: A Parameterized Non-linear Function Generator for Machine Learning Applications in FPGAs

Sam M.H. Ho and Hayden Kwok-Hay So
Department of Electrical and Electronic Engineering
The University of Hong Kong, Hong Kong
Email: {mhho, hso}@eee.hku.hk

Abstract—**Efficient implementation of machine learning applications on FPGAs often requires non-linear numerical functions with a non-standard numerical precision that is not readily available from vendor provided standard libraries. While application-specific designs of such functions can result in superior numerical accuracy and area efficiency when compared to ad-hoc composition using vendor-provided primitives, the effort devoted to this challenging task can hardly be portable to other similar applications. In this work, we present an open source generator, NnCore, for floating-point non-linear operator cores built using fixed-point piecewise polynomial segments. The proposed framework takes advantage of properties such as oddness/evenness and intercept-at-origin, often found in the numerical functions commonly used in machine learning applications, and applied an improved segmentation algorithm that specifically handles "outlier" segments, to reduce the required memory size for storing polynomial coefficients. Experimental results show that, at single-precision setting, NnCore generated cores use up to 65% fewer BRAMs, 63% fewer shift-registers, and runs at up to 2.2× the clock speed, compare with cores generated from a previous generic function generator. At half-precision, cores can run around 1.2× higher clock speed while requiring higher resource usage, or use a comparable number of resource but run at 12% to 45% lower clock speed. The use of HLS C++ as output format allows core integration into modern high-level workflow such as Xilinx SDAccel.**

I. Introduction

Recent years have witnessed a tremendous growth in interest to offload deep convolutional neural network inference and training on FPGA based custom computing machines for performance and power-efficiency reasons. Despite this widespread interest and the relatively regular nature of this application domain, however, efforts that would have allowed researchers to efficiently share, reuse and compare their results remain limited. Very often, researchers find themselves reinventing common features in hardware, such as a sigmoid activation function for a neural network, even when a very similar design has already been implemented in a related work. The need to reinvent may be a result of different requirements in data precision, function accuracy, or performance, while in many cases, it was merely because the source of the related work is not readily available. This lack of design reuse has a significant impact on the designers' productivity and greatly hinders further innovation in the use of custom computing machines in this application domain, especially when application specific designs are needed to fully exploit the potential of custom computing machines.

This work was supported in part by the Research Grants Council of Hong Kong (Project GRF 17245716), and the Croucher Foundation (Croucher Innovation Award 2013).

This work aims to help bridge this gap by introducing an open-source [1] floating-point activation function generator that is parameterizable and extensible. The proposed framework, NnCore, is capable of generating a family of non-linear functions that are commonly used as activation functions in a range of neural network training and inference applications, and can be extended to incorporate custom functions with relative ease. Not only are these functions commonly employed in many applications, as complex non-linear functions, they are also not immediately trivial for efficient implementation in hardware. A common framework thus facilitates optimised hardware implementation of these functions be easily reused and retargeted for different applications.

From a technical point of view, the proposed framework constructs floating-point operators using piecewise minimax-polynomials with fixed-point coefficients. To accommodate different application requirements, the framework allows users to make a trade-off between resource consumption and operator accuracy by specifying the maximum degree of the underlying polynomial, and the bit-width of the operator as input. It is, therefore, able to generate cores with standard half/single-precision, or custom-precision with variable non-standard mantissa and exponent width, which standard commercial tools do not support but are important to develop custom-made machine learning computing systems. In the current version, the proposed framework is capable of generating the hyperbolic tangent (tanh) and its derivative (d'tanh), sigmoid and its derivative (d'sigmoid), arc tangent (atan) and its derivative (d'atan), rectified linear unit (ReLU) and its variant ReLU6.

We thus consider the main contributions of this work is in the following areas:

- We have developed an open-source floating-point core generator for neural network activation functions and their derivatives, which is more optimised for the target function category than previous generic function generators;

TABLE I
AVAILABILITY OF VARIOUS MATH FUNCTIONS FROM VIVADO HLS

	Data Type	Avail.	Acc.(ULP)
atan	half	Yes, since 2017.1	n/a
	single/double	Yes	2
	custom	No	–
cosh	half/double	Yes, since 2017.1	n/a
	single	Yes	4
	custom	No	–
exp	half	Yes, since 2017.1	n/a
	single/double	Yes	Exact
	custom	No	–
sigmoid	–	No	–
sinh	half/double	Yes, since 2017.1	n/a
	single	Yes	6
	custom	No	–
tanh	half/single/double	Yes, since 2017.1	n/a
	custom	No	–
d'atan	–	No	–
d'sigmoid	–	No	–
d'tanh	–	No	–

- The proposed generator provides HLS C++ outputs, allowing more flexible resource/frequency trade-off during logic synthesis, and seamless integration into modern high-level tool-chain such as Xilinx SDAccel;
- We demonstrate the flexibility of the proposed framework in producing high-quality non-linear arithmetic cores with custom non-standard data precision for use in custom computing machines, which common commercial tools fail to achieve. Generated operators are faithfully rounded.

In the next section, related work in FPGA implementation for neural network will first be introduced. The design of the proposed framework will then be detailed in Section III, Section IV and Section V. Experimental results that evaluate the quality of the generated operators will be shown in Section VI. At last, we conclude this work in Section VII.

II. BACKGROUND

Part of the purpose of this work is to fill in the empty IP core space where factory provided IP cores do not exist for certain target functions, or the accuracy or flexibility of such IP cores do not meet the application's requirement. Table I illustrates the availability of some math cores from Xilinx.

For most functions, half-precision were only introduced since version 2017.1 of Vivado HLS, where no accuracy information was given in the documentation [2]. Accuracy information provided in the table are hence obtained from the 2016.4 version documentation [3]. This also reflects another problem, where users only with a license for older toolchains would not be able to use the newly added cores.

It can be seen from the table that for all of the functions included, only standard half/single/double-precision are supported, if available. No custom data-width is supported. Also, the accuracy differs between different operations. if a user is not aware of such difference while composing more complex operations, e.g. composing $tanh(x)$ from $sinh(x)/cosh(x)$ instead of $exp(x)$, the composed operator may suffer from suboptimal accuracy and resource usage, which we will show in section VI.

Since the activation functions' implementation is not the focus for many neural network application research works, in these works the activation functions tend to be approximated very roughly, e.g. piecewise linear approximate with number of segments from empirical experience [4][5][6][7], without quantifying the consequence on the networks' accuracy. Some [8] also apply segmented minimax polynomial, which gives better accuracy.

On the other hand, as there is a long history of research on neural networks hardware, there are a large amount of previous works on implementing the individual activation functions in hardware, particularly $tanh(x)$ and $sigmoid(x)$ as these were used since the early age of neural networks [9][10][11][12][13]. However, these works tend to focus on the resource reduction of a single function base of piecewise linear methods, compromising on the function's accuracy.

Works on more generic function evaluation, which is more similar to this work, includes [14], [15] and [16].

In [14] a hierarchical segmentation method was proposed to produce faithfully rounded segments, in an effort to reduce table size compared to uniform segmentation approaches. Fixed-point polynomials were used for evaluation, same as in this work. However, the method targets fixed-point inputs and was applied only to restricted input domains in the benchmarks with degree-1 and degree-2 polynomials.

The work [15] and [16] used similar binade partitioning approach and fixed-point polynomial approximation. However, in [15] function accuracy was not a focus, and an acceptable absolute error was used as one of the inputs to the generator. It also did not support multi-modal functions generation.

In this work, we based our algorithm on the work of [16], but added optimisations that are applicable to the category of activation functions used in neural networks, which we will discuss in section IV.

III. POLYNOMIAL APPROXIMATION OF ACTIVATION FUNCTIONS

A. Function Evaluation by Polynomial Approximation

In general, function evaluation, usually studied for elementary transcendental functions for implementing math libraries, includes the three steps: 1) range reduction, 2) evaluation, and 3) reconstruction. This is a well-studied topic with a long history and rich amount of literature. Readers are referred to materials like Muller's book [17] for more information. Here we only give a brief introduction.

Since it is hard to approximate a function over a large range at a high accuracy, range reduction is usually used so that the approximation would only need to be done in a relatively small range. There are two kinds of reduction, including additive reduction, where the reduced argument x^* is equal to $x - kC$, k being integer and C a constant, and the multiplicative reduction, where x^* equals x/C^k.

As an example, to implement the exponential function, Tang [18] used the additive reduction with $C = ln(2)/32$, such that the reduced argument x^* has a small range of $[-ln(2)/64, +ln(2)/64]$.

The reconstruction step is to deduce $f(x)$ back from $f(x^*)$. Take the above example, after we have an approximation $p(r)$ of e^{x^*}, we then construct

$$\begin{aligned} e^x &= e^{x^*+k\cdot ln(2)/32} \\ &= 2^{k/32}\cdot p(r) \end{aligned}$$

This is a simplified version of what was given in [18], but illustrates the idea of reconstruction.

The core of the three steps, the "evaluation" can be done by using polynomial (or rational) approximations, with or without tabulation, or by using shift-and-add algorithms like CORDIC.

In terms of hardware implementations, CORDIC based methods are usually iterative, requires less resource usage at the cost of performance. Polynomial estimations usually provide a higher throughput of 1 output per cycle, or at a configurable cycle-per-operation if resource sharing is built into the design.

The simplest form of a polynomial approximation that most people would have learnt is the Taylor series, but it does not give good approximations since it only approximates the target function around a particular point x. In fact, due to a theory by Chebychev, a special polynomial that minimizes the maximal error for approximation exist, known as the minimax polynomial. The Remez Exchange Algorithm is used to find minimax polynomials and is available in math packages like Mathematica and Maple.

However, it should be noted that, there is no guarantee on polynomial approximation being the most accurate, and that for some functions in some domains, a rational approximation can be superior over minimax approximation, like for the $\sqrt{x}$ function.

B. Activation Functions

Traditionally, the hyperbolic tangent and the sigmoid function, defined respectively as

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

and

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

have been used as activation functions in neural networks. They share the feature of having a limited output range, at $[-1, 1]$ and $[0, 1]$ respectively. The arc tangent function $tan^{-1}(x)$ also have a similar curve and serve as an activation function.

Recent works in the image processing area tends to use the ReLU function, especially in CNNs,

$$ReLU(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

otherwise, there is no general rule, other than the experience of previous works, on which activation functions should be used in what applications. A variation of the ReLU, the ReLU6, which equals $min(max(x, 0), 6)$, is commonly found in neural network packages like TensorFlow [19] and Torch [20], and is also available in our generator.

There are also relatively new activation functions being proposed like the Maxout [21] (which is not a simple math function that takes a single numeric input, but more like an extra layer) and the ELU [22], which were shown to perform well in various applications.

Other than the activation functions, when training a network using error back propagation, the derivatives of these activation functions are needed, and so a processor hardware for the training task will also need to have them implemented. The 1st order derivative of the hyperbolic tangent is

$$tanh'(x) = 1 - tanh^2(x)$$

, and that for sigmoid is

$$sigmoid'(x) = sigmoid(x)(1 - sigmoid(x))$$

One can see that these derivatives are complex functions made up of the original activation functions, and so an increase in resource usage is expected when implemented by assembling from multiple elementary functions.

From these formulations of activation functions and the respective derivatives, one should be able to see that range reduction is hard to apply since there is no obvious way to do reconstruction.

IV. Segment Search Algorithm

A. Terminologies

Here we introduce some terminologies before we go further into our algorithm. The floating-point numbers format used follows that of FloPoCo [23], where the first 2-bits represents the special numbers ($\pm 0, \pm\infty$, NaN), and regular numbers are of the form:

$$s \times 2^e \times (1 + f) \text{ where } 0 \leq f < 1, s \in \{-1, 1\}$$

and the bit of one in front of the significand is implicit, as in IEEE floating-point numbers. We also refer to the binade as the region of floating-point numbers that have the same exponent value:

$$bin(x) = (sign(x), \quad expnt(x))$$

The input domain is to be divided into segments S_i, each of which can be identified by its lower- and upper-bounds $\lfloor S_i \rfloor$ and $\lceil S_i \rceil$.

B. Algorithm

In this work, we implemented an algorithm similar to Thomas's work [16], and applied various optimisations that are feasible for our target functions. The basic idea behind the algorithm is to divide the input domain into segments base on the input and output binade, such that each resulting segment has a fixed input- and output- exponent pair, and so fixed-point polynomial approximation can be applied on the significand.

Our overall approach is as follows:

1) Make single-binade segments base on the input domain.
2) Walk along segment domain and split segments if their output crosses a binade.
3) Merge segments that have constant outputs of the same value.
4) Split segments until they can be approximated by a polynomial of degree d.
5) Split segments until they can be approximated by a fixed-point polynomial of degree at most d.
6) Detect and split "outlier" segments that have larger-than-average polynomial coefficients recursively.
7) Merge neighbouring segments if they have the same input-output binade, and are given the same approximation polynomial.
8) Merge neighbouring segments that are assigned the identity function, e.g. $y = x$, for approximation.

In the above flow, steps 3, 6, 7 and 8 are our optimisations methods, while steps 1, 2, 4 and 5 are mostly adapted from [16]. In step 1 we took advantage of the fact that many of our target functions are odd or even, so we only need to build the segments for the positive input domain in those cases, and map the sign bit of the output in the generated hardware accordingly.

In the original work of [16], after step 1 segments are also split according to their monotonicity, but since our target functions are mostly monotonic after taking advantage of their odd/even properties, we did not adapt this splitting.

In step 2), we walk along the segment domain, check if the output binade of the boundaries of segment S are equal, e.g.,

$$bin(f(\lfloor S \rfloor)) \equiv bin(f(\lceil S \rceil))$$

and split the segment if they are not. A binary search is applied here to find the maximum x where

$$r \leftarrow max\{x : \lfloor S \rfloor \leq x \leq \lceil S \rceil \wedge bin(f(x)) \equiv bin(\lfloor S \rfloor)\}$$

is set as the upper bound of the newly split segment.

We introduced step 3) according to the properties of our target functions. It is observed that many of the functions described approaches a limit at the ends of the input domain, and so a constant output can be set for a certain range of x. We can merge the already-split segments, as long as they can share the same output sign, exponent and the polynomial coefficients in the lookup table, in this case being a single constant value. This also saves runtime for the following steps of polynomial approximations.

After these steps, the resulting segments are all flat in both their domain and image, and we are ready to run fixed-point approximations on them. However, the runtime will be very slow if we do this directly at this stage, so approximations with real-coefficient polynomials are applied in step 4.

Given a segment S, a transformation on the target function f_t is applied, such that the transformed function maps from input significand to output significand, i.e. $f_s : [0, 1) \mapsto [0, 1)$:

$$f_s(x) = s_r \times 2^{-e_r} \times f_t(s_d \times 2^{e_d} \times (1 + x)) - 1$$

where $(s_d, e_d) = bin(\lfloor S \rfloor)$, $(s_r, e_r) = bin(f_t(\lfloor S \rfloor))$. This transformed function is then used as the approximation target for that segment.

Segments which have an approximation error $err_{approx} > 2^{-\omega_F - 3}$ is split, where ω_F is the width of the significand, until all segments can be approximated with such error bound. This target serves as a heuristic to pre-split more segments, such that the fixed-point approximation in the next step will be easier. Since the purpose of this step is to speed up the overall runtime, we only perform the approximation at the maximum degree constraint by the user. The `remez` command from Sollya [24] is used for this step, and the `dirtyinfnorm` command for quick error estimation.

Then at step 5, we try to approximate the remaining segments with fixed-point polynomials, as shown in the algorithm below. The while loop on line 5 serves as a heuristic, such that a lower-powered polynomial has a higher priority to be chosen first if it can reach the approximation error requirement. The checking on line 14 induces another splitting of segments, in the case where no polynomials were able to reach the error constraint. Although not shown here, the fixapprox method on line 6 also takes as input a list of formats, which is the number of fractional bits in the coefficients of the requested fixed-point minimax polynomial. The `fpminimax` and the `supnorm` command of Sollya are used for finding the fixed-point polynomial.

Algorithm 1 SegByFixPoly

```
1:  while $S_4 \neq \emptyset$ do
2:      $S \leftarrow \{s | s \in S_4\}$
3:      $S_4 \leftarrow S_4 / S$
4:      $i \leftarrow 0$
5:      while $i \leq d \wedge S \neq \emptyset$ do
6:          $(poly, err) \leftarrow fixapprox(f_s, S, i)$
7:          if $err \leq 2^{-\omega_F - 2}$ then
8:              $S_5 \leftarrow S_5 \cup \{(S, poly, err)\}$
9:              $S \leftarrow \emptyset$
10:         else
11:             $i \leftarrow i + 1$
12:         end if
13:     end while
14:     if $S \neq \emptyset$ then
15:         $m \leftarrow round((\lfloor S \rfloor + \lceil S \rceil)/2)$
16:         $S_4 \leftarrow S_4 \cup \{\{\lfloor S \rfloor..m\}\}$
17:         $S_4 \leftarrow S_4 \cup \{\{up(m)..\lceil S \rceil\}\}$
18:     end if
19: end while
```

Step 6 of the flow is one of the most critical steps we proposed, for generating polynomials that requires a reasonable amount of memory for the coefficients lookup table. During the development process, we found that after finishing the first 5 steps, the fixed-point polynomial of some resulting segments may contain coefficient of very large value above the integer part. This, in turn, requires significantly wider bit-width in each row in the coefficient ROM. We suspect the reason behind this is that the said segment may contain a very short sub-segment that is difficult to be approximated. This leads to the remaining parts of the segment being approximated with such a polynomial altogether. We call these segment “outliers”, and our solution for this is to filter every segment after step 5, and recursively split these outlier segments, whose polynomial coefficients have a maximum width of integer part that is two times larger than the average width.

In step 7, we try with least effort to merge any neighbouring segments that have the same binade and the same polynomial assigned. We do this because the binary splitting at steps 4, 5 and 6 may well be too pessimistic, since the boundary created from splitting a segment will never be removed once committed. But this is necessary for a fast runtime and good user experience, and so in this step, we pay the minimum effort trying to rescue some of these obvious cases.

In step 8 we perform an “identity function merging”, as we see that for functions that pass through the origin, lots of segments around $x = 0$ would be assigned the approximation $y = x$ when x is very small. Since the segments that will most likely be merged due to this technique is often clustered around 0, usually only one resulting segment is left on the lookup-table, and so we can hard-code these address information into the control logic of the polynomial evaluator, without increasing the bits to be stored in the coefficient lookup-table.

Table II shows the number of segments remaining after each of the steps 5 (”SegByFixPoly”), 6 (”SplitOutliers”), 7 (”MergeByPoly”) and 8 (”MergeByX”). Table III shows the average and the maximum number of integer bits in the fixed-point coefficients of all polynomial segments before and after step 6 (”SplitOutliers”).

From table II it is clear that the steps ”MergeByPoly” and ”MergeByX” are effective only for functions that pass through the origin $y = x = 0$, which is as expected. The ”MergeByX” step is capable of reducing segment counts by up to 35.8%, which is quite significant.

Looking at both tables it can be seen that except for the sigmoid function at half-precision and degree 4, the ”SplitOutliers” step only increased the number of segments by less than 12%, and reduced maximum coefficient bit-width of integer part by up to 100%. This is important because the polynomial coefficients are stored in ROM, with each segment occupying for one row, and the bit-width decided by the one segment with the highest power of polynomial and widest coefficient. Reducing this maximum integer bit-width allow all segments to be stored in a ROM with narrower bit-width.

V. Core designs

Our floating-point function approximator is similar to that in figure 1 of [16], consisting of ”segmentation”, ”table-lookup” and ”fixed-point polynomial evaluation” stages. The difference in ours, in particular, is that the evaluator reads also the address to be used for table-lookup, compares its value against a number of compile-time hard-coded values, to determine whether the evaluation can be skipped, as mentioned in step 8 of section IV.

TABLE II
Number of segments after each algorithm stage

Func.	Impl.	S5	SplitOutliers		MergeByPoly		MergeByX	
		segs.	segs.	%i	segs.	%d	segs.	%d
tanh	wf10d3	61	66	8.2	65	1.5	57	12.3
	wf10d4	52	58	11.5	57	1.7	49	14.0
	wf23d3	358	368	2.8	364	1.1	250	31.3
	wf23d4	307	322	4.9	318	1.2	204	35.8
d’tanh	wf10d3	49	49	0	49	0	49	0
	wf10d4	36	40	11.1	40	0	40	0
	wf23d3	2044	2044	0	2044	0	2044	0
	wf23d4	542	542	0	542	0	542	0
atan	wf10d3	67	72	7.5	71	1.4	63	11.3
	wf10d4	62	69	11.3	68	1.4	60	11.8
	wf23d3	448	459	2.5	455	0.9	341	25.1
	wf23d4	357	373	4.5	369	1.1	255	30.9
d’atan	wf10d3	67	70	4.5	70	0	70	0
	wf10d4	59	62	5.1	62	0	62	0
	wf23d3	2937	2945	0.3	2945	0	2945	0
	wf23d4	1160	1170	0.9	1170	0	1170	0
sigmoid	wf10d3	69	77	11.6	77	0	77	0
	wf10d4	57	124	117.5	124	0	124	0
	wf23d3	2109	2109	0	2109	0	2109	0
	wf23d4	598	598	0	598	0	598	0
d’sigmoid	wf10d3	45	45	0	45	0	45	0
	wf10d4	34	36	5.9	36	0	36	0
	wf23d3	2013	2013	0	2013	0	2013	0
	wf23d4	535	535	0	535	0	535	0

TABLE III
Maximum no. of bits in the integer part of all fixed-point coefficents in all segments

Func.	Impl.	SegByFixPoly		SplitOutliers	
		avg p	max p	max p	%d
tanh	wf10d3	1	9	0	100
	wf10d4	2	22	4	81.8
	wf23d3	1	38	1	97.4
	wf23d4	2	72	4	94.4
d’tanh	wf10d3	4	7	7	0
	wf10d4	3	9	7	22.2
	wf23d3	13	16	16	0
	wf23d4	16	20	20	0
atan	wf10d3	1	9	1	88.9
	wf10d4	1	17	3	82.4
	wf23d3	1	38	2	94.7
	wf23d4	1	72	3	95.8
d’atan	wf10d3	2	14	2	85.7
	wf10d4	2	14	2	85.7
	wf23d3	3	37	4	89.2
	wf23d4	3	37	6	83.8
sigmoid	wf10d3	2	7	5	28.6
	wf10d4	2	8	4	50
	wf23d3	13	16	16	0
	wf23d4	14	20	20	0
d’sigmoid	wf10d3	3	7	7	0
	wf10d4	3	8	7	12.5
	wf23d3	13	16	16	0
	wf23d4	16	20	20	0

The segmentation module takes the whole floating-point number as input, and returns the address of the corresponding segment in the coefficient lookup table. The module is built with $k = \lceil log_2(n) \rceil$ comparators, each compares the input x with a segment boundary read from register, effectively performing a binary segment search, with each stage recovering one bit of the segment address output. It also checks the boundary value against hard-coded maximum, such that it can accept any segment count n.

The bit width in the LSBs of coefficients in the ROM

TABLE IV
RESOURCE USAGE OF RELU

	Prec.	LUT	FF	BRAM	DSP
ReLU	5,10	9	0	0	0
	8,23	17	0	0	0
ReLU6	5,10	19	0	0	0
	8,23	35	0	0	0

follows the heuristic suggested in [16], which is:

$$l_i = 2^{-\omega_f+i-1}$$

To achieve faithful rounding, the requirement would be:

$$\epsilon_{approx} + \epsilon_{eval} + \epsilon_{round} \leq 2^{-\omega_f}$$

Since we set the approximation error bound in line 7 of algorithm 1 to be $2^{-\omega_f-2}$, and the final rounding error is $2^{-\omega_f-1}$, our evaluation error budget would be $2^{-\omega_f-1}$. To satisfy this goal, the bit width of the output of each stage of the Horner form evaluation is set to:

$$\begin{aligned} a_d &= c_d \\ a_i &= round(c_i + x \times a_{i+1}, \quad 2^{-\omega_f-i-g}) \\ a_0 &= round(c_0 + x \times a_1, \quad 2^{-\omega_f}) \end{aligned}$$

where $round(x, r)$, and g is a predetermined number of guard bits to achieve faithful rounding.

Currently, generated operators are output in Xilinx HLS C++ form factor, for the ease of integrating the cores into other projects as IPs, e.g. [25]. However, there's no limit in the generator for supporting other output formats, e.g. to use FloPoCo operators for the fixed-point evaluator. Along with the generated core, we also generate the corresponding test vector file. A C++ test bench was written to take these test vector files and verify against the generated cores for correctness in code generation. The operators are faithfully rounded (ULP = 1) by construction, so testing for accuracy is not the main purpose of this test bench.

VI. EXPERIMENT RESULTS

In our experiment setups, the target device was set to the Alpha-Data ADM-PCIE-7V3 board, featuring a xc7vx690tffg1157-2 chip from Xilinx.

Unlike other generated functions, the ReLU and ReLU6 are generated from our older code base published in [26]. Generated codes are in Verilog RTL format, and the post-placement and route results are shown here in table IV for the sake of completeness.

The tool version used in the experiments was Xilinx Vivado HLS 2016.2. NnCore is written in Python 3, while all the math operations used within the algorithm were passed to Sollya 6.0 [24] for processing. Due to the lack of license for the newer version of the Vivado tool (2017.1 and up), we were not able to provide benchmark results for the newer half-precision operators introduced since version 2017.1, as listed in table I of section II. Hence, other than the $atan(x)$ function which is available from the tool, other HLS implementations used as a baseline for comparisons were composted using primitive operators as follows.

$tanh(x)$ is given by either $(e^x - e^{-x})/(e^x + e^{-x})$, or $sinh(x)/cosh(x)$. It's derivative given by $\mathrm{d}tanh/\mathrm{d}x = 1 - tanh^2(x)$; $\mathrm{d}atan(x)/\mathrm{d}x = 1/(1 + x^2)$; $sigmoid(x) = 1/(1 + e^{-x})$ and $\mathrm{d}sigmoid(x)/\mathrm{d}x = e^{-x}/(1 + e^{-x})^2$.

Since part of our algorithm is based on the work of [16], comparison with the operators generated by it is also provided, referred to as "FloatApprox" in the tables. The source code was downloaded from an obscure branch of FloPoCo [23] named "random", at [27]. Since the version of FloPoCo on that branch was marked as 2.5.0, we followed the installation instruction of FloPoCo 2.5.0 and used FPLLL 3.0.12 library and Sollya 3.0 as dependencies.

Resource usage and clock achieved after out-of-context placement and route flow is presented. To find the maximum achievable clock, we set an initial target clock period, decrease the clock period at a 0.1ns step and re-iterate starting from high-level synthesis, until the post-route timing target was not met. The result with the highest clock speed achieved is reported.

For the operators generated by "FloatApprox", which were in VHDL RTL, we extracted the auto-generated out-of-context synthesis and placement-and-route scripts from the HLS projects, and applied to the generated VHDL.

Typically, generated cores give more all-rounded results on both resource usage, design latency and clock speed when the max power of polynomial is set to 3 or 4, hence results of only these settings are provided in the experiments, but such assumption may not hold true for every target functions. Table V shows the design latency and post-placement and route results of generated functions in single-precision. Numbers in brackets are the results normalised by those of the reference HLS design. The reader should bear in mind that, both "Float-Approx" and "NnCore" generated cores are faithfully rounded (ULP = 1), while the reference composed cores mostly have lower accuracies, so it would be natural if generated cores involve higher resource usage.

Note that for the single-precision configuration, "FloatApprox" failed to generate a design in 8 out of 12 setups, which is significantly worse than the original results presented in [16]. We tried our best effort to reach the author of [16], to see if any specific dependencies might be needed, but without success as of the time of writing. Yet we consider the results presented here valid, since anyone following the same steps as us would produce the same results, if no additional knowledge on how to replicating the results in [16] is given.

From Table V we can see that composing the $tanh(x)$ with $sinh(x)/cosh(x)$ could involve more than $2\times$ the resource required compare to using $exp(x)$.

For $tanh(x)$, $d'tanh(x)$ and $atan(x)$, both "NnCore" and "FloatApprox" generated designs use fewer slices, LUTs and flip-flops than references, but more DSPs for $atan(x)$ and more BRAMs for all 3.

For $d'atan(x)$, $sigmoid(x)$ and $d'sigmoid(x)$, slices and LUTs used are about under $2\times$ and flip-flops about $1\times$, with

TABLE V
POST-PLACE AND ROUTE RESULTS FOR SINGLE-PRECISION OPERATORS IN VIRTEX-7

Func.	Impl.	Spec.	Lat.		Slice		LUT		FF		DSP		BRAM	SRL		Clk.	
tanh	HLS comp.	exp	75	–	1098	–	2955	–	4123	–	14	–	0	143	–	517.9	–
		sinh, cosh	117	(1.56)	2841	(2.59)	9210	(3.12)	8953	(2.17)	30	(2.14)	0	233	(1.63)	342.7	(0.66)
	NnCore	d = 3	63	(0.84)	555	(0.51)	1082	(0.37)	2282	(0.55)	5	(0.36)	4	243	(1.70)	359.6	(0.69)
		d = 4	84	(1.12)	702	(0.64)	1109	(0.38)	3061	(0.74)	8	(0.57)	3	264	(1.85)	566.6	(1.09)
d'tanh	HLS comp.	exp	97	–	1284	–	3342	–	4894	–	17	–	0	159	–	501.5	–
	NnCore	d = 3	66	(0.68)	495	(0.39)	1076	(0.32)	1808	(0.37)	8	(0.47)	28	350	(2.20)	337.6	(0.67)
		d = 4	88	(0.91)	955	(0.74)	2740	(0.82)	2553	(0.52)	12	(0.71)	8	330	(2.08)	458.5	(0.91)
atan	HLS synth.	–	124	–	2817	–	9414	–	8979	–	2	–	0	413	–	370.2	–
	FloatApprox	d = 3	40	(0.32)	562	(0.20)	1471	(0.16)	1590	(0.18)	6	(3.00)	3	362	(0.88)	244.4	(0.66)
		d = 4	49	(0.40)	640	(0.23)	1753	(0.19)	1904	(0.21)	8	(4.00)	4	542	(1.31)	247.8	(0.67)
	NnCore	d = 3	79	(0.64)	758	(0.27)	1486	(0.16)	3065	(0.34)	6	(3.00)	2	270	(0.65)	554.3	(1.50)
		d = 4	56	(0.45)	595	(0.21)	957	(0.10)	2677	(0.30)	7	(3.50)	4	203	(0.49)	393.1	(1.06)
d'atan	HLS comp.	mul, div	51	–	479	–	1178	–	2195	–	3	–	0	53	–	548.5	–
	FloatApprox	d = 3	43	(0.84)	709	(1.48)	1918	(1.63)	1953	(0.89)	6	(2.00)	32	373	(7.04)	230.9	(0.42)
		d = 4	51	(1.00)	679	(1.42)	1604	(1.36)	2295	(1.05)	8	(2.67)	22	576	(10.87)	232.1	(0.42)
	NnCore	d = 3	66	(1.29)	1911	(3.99)	6466	(5.49)	2098	(0.96)	6	(2.00)	11	358	(6.75)	402.1	(0.73)
		d = 4	90	(1.76)	802	(1.67)	2143	(1.82)	2159	(0.98)	8	(2.67)	19	357	(6.74)	468.6	(0.85)
sigmoid	HLS comp.	exp	75	–	722	–	1840	–	2771	–	7	–	0	90	–	511.5	–
	NnCore	d = 3	88	(1.17)	1373	(1.90)	3812	(2.07)	2978	(1.07)	9	(1.29)	30	367	(4.08)	419.1	(0.82)
		d = 4	89	(1.19)	944	(1.31)	2423	(1.32)	3069	(1.11)	12	(1.71)	11	334	(3.71)	454.3	(0.89)
d'sigmoid	HLS comp.	exp	83	–	764	–	1962	–	2986	–	10	–	0	121	–	523.6	–
	NnCore	d = 3	83	(1)	942	(1.23)	2338	(1.19)	2657	(0.89)	9	(0.90)	23	358	(2.96)	467.3	(0.89)
		d = 4	88	(1.06)	795	(1.04)	1884	(0.96)	2426	(0.81)	12	(1.20)	13	335	(2.77)	465.3	(0.89)

the exception of $d'atan(x)$ at degree = 3 for "NnCore".

In all functions except $atan(x)$, generated cores uses between 1.7× to 6.75× the shift-registers for "NnCore", between 1.63× to 10.87× for "FloatApprox". On achievable clock frequencies, most "NnCore" generated designs run at about 0.7× to 1.5× the clock of reference designs.

Compare with "FloatApprox", "NnCore" generated designs use up to 65% fewer BRAMs, 63% fewer shift-registers, and runs at up to 2.2× the clock speed at the same setting. The clock speed advantage may be due to the fact that "NnCore" generates HLS C++ code as output, enjoying better optimisations during high-level synthesis.

Table VI shows the comparison between "FloatApprox" and "NnCore" generated functions at half-precision setting. As mentioned above since no composite implementation is possible before Vivado HLS 2017.1, no reference implementation is provided. In general, "NnCore" generated cores required more resources than that of "FloatApprox" except for DSPs, while being able to run at about 1.2× higher clock speed.

The higher resource usage is partly due to that when we iterate for the highest clock achievable, we started from the high-level synthesis step, which in turn tries to generate faster circuits at a compromise of resource usage. To prove this, we also provide a setup named "NnCore-min" in table VI, where we set the target clock to 250MHz and report the actual clock and resource usage achieved.

At this setting "NnCore" generated cores use comparable number of slices and LUTs to "FloatApprox" cores, up to 30% fewer number of shift-registers in most cases, about 2× the number of flip-flops in $tanh(x)$, $atan(x)$ and $sigmoid(x)$, or otherwise similar amount of flip-flops in $d'tanh(x)$, $d'atan(x)$ and $d'sigmoid(x)$. The clock speed achieved range from 12% to 45% lower than that of "FloatApprox" cores.

In addition to the latency and resource usage metrics same as in table V, here we also listed the generators' runtime on our machine with an Intel Core i7-3820 CPU at 3.6GHz. For "NnCore-min", since re-run of core generations is not needed, generator runtime for these entries was skipped in the table.

Normally this metric is not relevant as long as the generator finishes in a reasonable amount of time, say within several hours. However, during our experiments, we found that the "FloatApprox" sometimes required an extensive amount of time to run. For the functions $tanh(x)$ and $d'tanh(x)$, it took up to 2.5 days to produce the generated operators. This, together with the issue of failing to generate most cores at single-precision, seriously hindered the use of the "FloatApprox" generator. Comparatively, "NnCore" generated cores within a reasonable amount of time across all target functions.

VII. CONCLUSION

In this paper, we present NnCore, an open-source, parameterizable and extensible activation function generator for FPGA based machine learning applications. NnCore allows easy generation of non-linear functions that are common to many different machine learning applications, with a goal to further facilitate design reuse among the research community. NnCore constructs operators using piecewise fixed-point minimax-polynomials and allows users to specify the maximum polynomial power, bit widths of the exponent and the significand as parameters.

Experimental results show that NnCore is capable of generating functions with a comparable number of slices, LUTs and flip-flops usage, a fewer number of block RAMs and shift-registers, and significantly higher clock frequencies compare to a previous generic function generator. Generated cores are faithfully rounded (ULP = 1), which is not available from composited functions. NnCore also showed stability across all generation settings, which is not shown in the previous generator. The use of HLS C++ as output format also allows

TABLE VI
POST-PLACE AND ROUTE RESULTS FOR HALF-PRECISION OPERATORS IN VIRTEX-7. "FA" STANDS FOR "FLOATAPPROX; "NC" FOR "NNCORE"; "NCM" FOR "NNCORE-MIN".

FUNC	Impl.	DEG	Lat.	SLICE	LUT	FF	DSP	BRAM	SRL	Clk.	Runtime
tanh	FA	3	22	105	252	375	3	0	87	542.0	2434 m 1 s
		4	27	109	263	407	4	0	102	555.2	2476 m 16 s
	NC	3	56	274	433	1108	3	0	121	656.2	1 m 18 s
		4	62	315	508	1399	4	0	141	663.6	1 m 41 s
	NCm	3	26	157	263	716	3	0	63	479.2	–
		4	27	180	325	770	4	0	84	348.9	–
d'tanh	FA	3	22	130	302	487	3	0	108	541.7	3652 m 27 s
		4	27	135	334	522	4	0	140	561.2	3643 m 46 s
	NC	3	56	227	468	894	3	0	137	644.7	1 m 51 s
		4	66	278	573	1117	4	0	158	648.1	1 m 51 s
	NCm	3	24	145	329	529	3	0	79	345.1	–
		4	26	156	416	570	4	0	97	335.5	–
atan	FA	3	22	99	241	370	3	0	81	550.1	22 s
		4	27	113	270	419	4	0	108	554.6	40 s
	NC	3	53	252	431	1012	3	0	118	634.5	2 m 2 s
		4	60	295	504	1247	4	0	137	638.2	2 m 32 s
	NCm	3	26	157	277	719	3	0	68	470.1	–
		4	29	194	344	819	4	0	82	336.9	–
d'atan	FA	3	22	105	270	428	3	0	84	539.1	9 s
		4	27	132	320	492	4	0	124	544.7	13 s
	NC	3	59	218	454	815	3	0	135	644.3	17 m 37 s
		4	67	257	530	997	4	0	154	635.7	54 m 41 s
	NCm	3	26	148	327	527	3	0	85	362.2	–
		4	29	178	386	590	4	0	100	325.8	–
sigmoid	FA	3	22	99	210	328	3	0	76	578.7	30 s
		4	27	99	230	363	4	0	92	586.2	19 s
	NC	3	66	320	653	1225	3	0	157	634.1	2 m 33 s
		4	68	328	744	1258	4	0	165	628.5	3 m 23 s
	NCm	3	26	165	444	602	3	0	84	317.1	–
		4	32	199	470	668	4	2	110	334.9	–
d'sigmoid	FA	3	22	128	307	486	3	0	108	535.3	10 s
		4	27	133	341	519	4	0	140	525.8	1 m 0 s
	NC	3	53	199	418	833	3	0	133	644.7	6 m 7 s
		4	63	256	500	1035	4	0	147	634.5	7 m 27 s
	NCm	3	24	127	292	496	3	0	79	354.6	–
		4	26	146	324	533	4	0	95	337.6	–

integration of cores into modern high-level workflow such as Xilinx SDAccel, which is a unique feature.

REFERENCES

[1] S. M. Ho. NnCore repository. [Online]. Available: https://bitbucket.org/hku-casr/nncore

[2] Xilinx. (2017, Apr.) Vivado design suite user guide ug902 v2017.1. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-synthesis.pdf

[3] ——. (2016, Nov.) Vivado design suite user guide ug902 v2016.4. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug902-vivado-high-level-synthesis.pdf

[4] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 32–37.

[5] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 356–367.

[6] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2014.

[7] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 269–284. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541967

[8] J. C. Ferreira and J. Fonseca, "An FPGA implementation of a long short-term memory neural network," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Nov 2016, pp. 1–8.

[9] H. Faiedh, Z. Gafsi, and K. Besbes, "Digital hardware implementation of sigmoid function and its derivative for artificial neural networks," in *ICM 2001 Proceedings. The 13th International Conference on Microelectronics.*, Oct 2001, pp. 189–192.

[10] K. Basterretxea, J. M. Tarela, and I. del Campo, "Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons," *IEE Proceedings - Circuits, Devices and Systems*, vol. 151, no. 1, pp. 18–24, Feb 2004.

[11] C.-W. Lin and J.-S. Wang, "A digital circuit design of hyperbolic tangent sigmoid function for neural networks," in *2008 IEEE International Symposium on Circuits and Systems*, May 2008, pp. 856–859.

[12] B. Zamanlooy and M. Mirhassani, "Efficient VLSI implementation of neural networks with hyperbolic tangent activation function," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 1, pp. 39–48, Jan 2014.

[13] A. M. Abdelsalam, J. M. P. Langlois, and F. Cheriet, "A configurable FPGA implementation of the tanh function using DCT interpolation," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 168–171.

[14] D. U. Lee, R. C. C. Cheung, W. Luk, and J. D. Villasenor, "Hierarchical segmentation for hardware function evaluation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 1, pp. 103–116, Jan 2009.

[15] L. Ge, S. Chen, Y. Nakamura, and T. Yoshimura, "A synthesis method of general floating-point arithmetic units by aligned partition," *IPSJ Transactions on System LSI Design Methodology*, vol. 1, pp. 67–77, 2008.

[16] D. B. Thomas, "A general-purpose method for faithfully rounded floating-point function approximation in FPGAs," in *2015 IEEE 22nd Symposium on Computer Arithmetic*, June 2015, pp. 42–49.

[17] J. Muller, *Elementary Functions: Algorithms and Implementation*, ser. Computer Science. Birkhauser Boston, 2006. [Online]. Available: http://books.google.com/books?id=Mx-_kaANJBEC

[18] P. T. P. Tang, "Table-driven implementation of the expm1 function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software (TOMS)*, vol. 18, no. 2, pp. 211–222, 1992.

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *ArXiv e-prints*, Mar. 2016.

[20] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[21] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout Networks," *ArXiv e-prints*, Feb. 2013.

[22] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)," *ArXiv e-prints*, Nov. 2015.

[23] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[24] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An environment for the development of numerical codes," in *Mathematical Software - ICMS 2010*, ser. Lecture Notes in Computer Science, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327. Heidelberg, Germany: Springer, September 2010, pp. 28–31.

[25] S. M. H. Ho, M. Wang, H. C. Ng, and H. K. H. So, "Towards FPGA-assisted Spark: An SVM training acceleration case study," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Nov 2016, pp. 1–6.

[26] S. M. Ho, C.-H. D. Hung, H.-C. Ng, M. Wang, and H. K.-H. So, "A parameterizable activation function generator for FPGA-based neural network applications," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on.* IEEE, 2017, pp. 84–84.

[27] FloPoCo repository. [Online]. Available: https://scm.gforge.inria.fr/anonscm/git/flopoco/flopoco.git