Activation Function Architectures for FPGAs

Martin Langhammer
Intel UK

Bogdan Pasca
Intel France

***Abstract*—Machine Learning is now one of the most active application areas for FPGAs. The more complex recurrent neural network (RNN) topologies require multiple non-linear activation functions, mainly *tanh* and *sigmoid*, per iteration. In this paper we will examine the impact of activation function quality - in both area and (especially) latency - on RNN performance. We present a number of architectures for these functions, for both half precision (IEEE754-2008 FP16) and single precision (IEEE754 FP32) floating-point representations. We describe how the IEEE754 single precision hard floating point (HFP) blocks available in current FPGAs ease the implementation of these functions, and we also give an alternate method the *tanh* function based on integer arithmetic. With the combination of exceptional internal memory bandwidth, direct support of high performance floating point dot products, and the new activation functions, we show that FPGAs can be a highly effective vehicle for these type of neural networks.**

***Keywords*-FPGA; activation functions; hyperbolic tangent; sigmoid; machine learning;**

I. Introduction

Recursive neural networks such as LSTM typically make use of two activation functions: the hyperbolic tangent and the sigmoid. Many applications, especially for training RNNs, utilize floating point representations - typically single precision, although increasing half precision, or FP16 - even though some RNN inferencing is now implemented with low precision integer values [1]. Naive floating-point (FP) constructions of these activation functions assemble primitive operators, using well known identities (Eq. 8 and Eq. 13) involving other elementary functions; the operators include the FP exponential [2], [3] and reciprocal or division functions [4]. There are two main issues with operator assembly. First, despite efficient individual implementations of e^x and $1/x$, the area of the circuit can be high - we show in this paper that area can be greatly reduced if the activations function are implemented directly. Secondly, but more significantly, the two chained operations result in a long latency - this severely impacts the performance of the iterative RNN, and is out of proportion to the computational fraction of the activation functions compared to the dense matrix-vector multiplies.

This paper introduces several new floating-point architectures for the two activation functions. For *tanh* we show two classes of implementations: (a) architectures which are based on integer arithmetic, for use on devices without dedicated HFP blocks - Sections IV-A1 and IV-A2 (b) architectures for Hard FP (HFP) enabled FPGAs (such as the Intel Arria 10 [5] and Stratix 10 [6] devices) - Sections IV-B1 and IV-B2. For sigmoid we only describe architectures for HFP FPGAs - Sections V-A and V-B.

This work has two main contributions, each with multiple sub-parts.

(a) The adaptation of the algorithm from [7] to the hyperbolic tangent which includes:
   i. *tanh* specific analysis of the compute range
   ii. fixed-point argument decomposition that results in efficient and accurate architectures, and
   iii. an efficient reciprocal computation for this context.

(b) the use of floating-point piecewise polynomial approximation (PPA) for HFP based architectures, including:
   i. use of non-uniform and mixed-segmentation schemes for half-precision structures
   ii. use of odd, degree-5 polynomial for single precision (SP) *tanh* that provides a good balance between DSP and embedded memory (M20K) usage, and
   iii. modified polynomial argument generation for the sigmoid in order avoid catastrophic evaluation errors.

The paper is organized as follows. First, an introduction to RNN architectures and the impact of activation function quality will be analyzed in Section II. A short review of prior activation function results follows in Section III. The hyperbolic tangent implementation will be presented in Section IV, for both FP16 and FP32 using integer DSP Blocks, and then FP16 and FP32 cores using the HFP blocks. Similarly, the sigmoid function is presented for both FP16 and FP32 cases, in Section V, although only for HFP blocks. The results for both of these are analyzed in Section VI. We then finish with the conclusions and references.

II. RNN Architectures

A good introduction to RNNs is given in [8], from which Figure 1 and Equations 1-6 are taken. We will not delve into the algorithms, but restrict ourselves to the dataflow within a single iteration. Many iterations are used in an RNN application. Typically, a single iterative core is reused multiple times. A new iteration cannot start until the previous one has completed. In a parallel compute environment such as an FPGA, the latency through the iteration therefore has a direct impact on the performance of the RNN. By

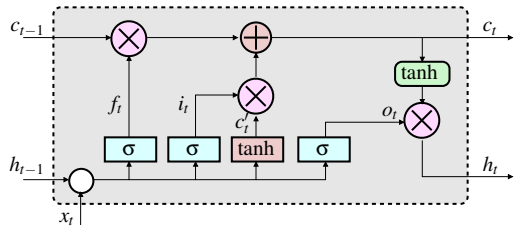


Figure 1. RNN node

inspection, almost every path contains an activation function. The equations 1-6 describe the operation at each node, and can be expanded out to Equation 7 to analyze the critical path through the core.

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \quad (1)$$

$$i_t = \sigma(W_f[h_{t-1}, x_t] + b_{fi}) \quad (2)$$

$$c'_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \quad (3)$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \quad (4)$$

$$c_t = f_t \times c_{t-1} + i_t \times \quad (5)$$

$$h_t = o_t \times \tanh(c_t) \quad (6)$$

$$h_t = o_t \times \tanh(f_t \times c_{t-1} + i_t \times \tanh(W_c[h_{t-1}, x_t] + b_c)) \quad (7)$$

We can see that the critical path contains one matrix-vector multiply, and two activation functions. From [9], the core of a matrix-vector multiply (a dot product) can be directly implemented in an FPGA with HFP capability. For an example vector dimension of 64, the latency of the dot product core is only around 20 cycles. If the activation functions are constructed by operator assembly [3], we can see that the combination of multiple operators will result in a latency many times greater than the dot product. From [3] we can see that a typical operator may contain 6 HFP blocks; the assembly of multiple operators will therefore still only contain a handful of HFP blocks, however, the latency of the activation function will be much greater than the much larger (in the number of computational terms) matrix-vector block. The latency of the two activation functions will therefore dominate the path through the core. This imbalance will give a powerful incentive to optimize the implementation of the activation functions.

To give a simple illustration, assume that the activation function via operator assembly has a latency of 50 cycles, and the matrix-vector dot product has a latency of 20 cycles. The two chained activation functions therefore contribute almost 80% of the latency of the core (there are some additional multipliers and adders in equation 7). Reducing the activation function latency by one half through developing a direct implementation of the *tanh* and *sigmoid* will therefore almost half the latency of each iteration.

III. Related work

There is very little work on the implementation of these activation functions in floating-point for FPGA architectures. In [10] a kernel processor is used to iteratively compute the values of the two functions. No details are given on how the two functions are broken down into instructions, or how the instructions are scheduled. In [11] an iterative architecture based on an expanded McLaurin series for the exponential is presented. The system comprises a FP adder, multiplier and divider together with a handcrafted FSM that allows computing either of the two functions. CORDIC-based activation function implementations are shown in [12] for 32 and 64-bit fixed point. No frequency or latency numbers are given, and it is not clear if the presented synthesis results are for 32 or 64-bits. In [13] Zhang presents synthesis results for a SP FP implementation of the sigmoid function.

IV. Hyperbolic tangent

The hyperbolic tangent is defined by the identity shown in Eq. 8.

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} = 1 - \frac{2}{e^{2x}+1} \quad (8)$$

Taylor's series for *tanh* in zero is $\tanh(x) = x - 1/3 \cdot x^3 + 2/15 \cdot x^5 - ....$ For a FP implementation, a good approximation is $\tanh(x) = x$ for $|x| < 2^{-\lceil \frac{wF}{2} \rceil}$ - here wF is the FP fraction width, with single-precision (SP) having $wF = 23$ and half-precision (HP) having $wF = 10$. We use this approximation for SP as $|x| < 2^{-12}$, and for HP as $|x| < 2^{-5}$.

The hyperbolic tangent is symmetric with respect to the origin $\tanh(-x) = -\tanh(x)$. For our implementation, we focus only on the positive interval, and reconstruct the negative range results by matching input and output signs.

From Eq. 8 we see that as $\frac{2}{e^{2x}+1}$ becomes smaller than $1/2ulp$ of 1, the subtraction returns 1. Here *ulp* stands for *unit in last place*, or in other words the weight of the LSB of the floating-point result [14].

$$\frac{2}{e^{2x}+1} < 2^{-wF-1} \rightarrow x > \frac{\log(2^{wF+2}-1)}{2} \quad (9)$$

We conclude that the hyperbolic tangent has a restricted input range for which the output varies. The bound given in Eq. 9 ensures an approximation with $1/2ulp$ accuracy. The compute interval can be further reduced by relaxing the bounds in this equation. Below we have a number of bounds for x:

precision	$<1/2ulp$	$<1ulp$	$<2ulp$
half (wF=10)	4.15	3.81	3.46
single (wF=23)	8.66	8.31	7.87

The following identity allows performing an additive range reduction over the argument of the hyperbolic tangent. A similar identity holds for the tangent, but the denominator shows a negative sign.

$$\tanh(a+b) = \frac{\tanh(a) + \tanh(b)}{1 + \tanh(a)\tanh(b)} \quad (10)$$

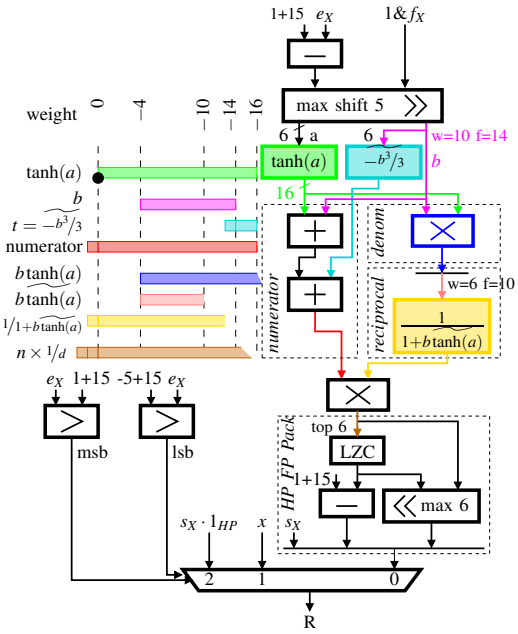


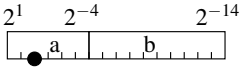
Figure 2. Half-precision $\tanh(x)$ architecture

A. Integer DSP Blocks Only

We present an implementation that extends the techniques described in [7] to the hyperbolic tangent.

1) Half Precision: We approximate $\tanh(x) \approx x$ for small values, $|x| < 2^{-4}$; the largest relative error for this approximation is approximately $1.25ulp$ when $|x| \approx 2^{-4}$. For $|x| \geq 4$ we approximate the function by the FP value $s_X \cdot 1_{HP}$, with a worst-case relative error less than $1ulp$.

The compute range is reduced to the interval $[2^{-4}, 4)$. An unsigned fixed-point format having width 16 and fraction 14 allows for a lossless representation of the HP values in this range. The alignment shifter will only shift by a maximum of 5 positions; inputs requiring right shifting beyond this value are handled by the two previously described branches. We use the additive range reduction from Eq. 10 with the values a and b selected as follows:



The numerator $\tanh(a) + \tanh(b)$ can be approximated by $\tanh(a) + b + t$. Here t is a 3-bit approximation of $-b^3/3$ obtained by table lookup indexed by the 6 MSBs of b. We use tilde variables for the approximations. The 16-bit value $\tanh(a)$ is obtained by tabulation; the largest approximation error is less than 2^{-17}. The error in term t has two components: (i) the initial truncation error (truncating b to its 6 MSBs) which has a low weight of $\approx 2^{-32}$, and (ii) the rounding error for storing this result on 3 bits; this error will be 2^{-17}. The sum of the three errors is loosely bounded by 2^{-16} which gives the total approximation and evaluation error for the numerator.

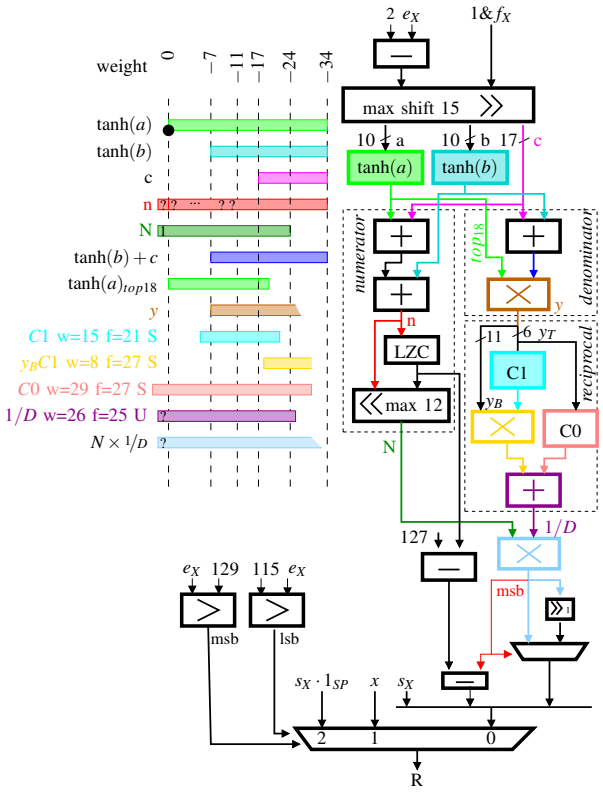


Figure 3. Single-precision $\tanh(x)$ architecture

The denominator is approximated by $1 + b\tanh(a)$. The approximation error is largest when $a = 11.1111_2$ and $b = 00.0000111111_2$, and is of the order 2^{-15}. Truncating $b\tanh(a)$ at weight -10 we are left with an approximation of the denominator accurate within $2^{-10} + 2^{-15}$. Since the MSB of term $b\tanh(a)$ is of the order 2^{-5}, we can use the 6 bits of $b\tanh(a)$ to index a table for the fixed-point reciprocal. The table will store a 13-bit result with a 12-bit fraction.

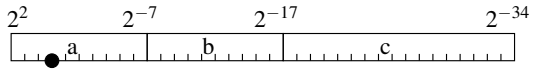
The product of $num \times recip(denom)$ is then normalized and the result exponent is created. The leading zero counter will only check the most significant 6 bits of the product, and the normalization shifter size is reduced since it may only left-shift by at most 6 positions. The full architecture is depicted in Figure 2 and the fixed-point alignments used for computing the numerator and denominator are shown on the left.

2) Single Precision: For SP we approximate $\tanh(x) \approx x$ for $|x| < 2^{-11}$, and $\tanh(x) = 1$ for $|x| \geq 8$. The approximation error is largest for $|x| \approx 8$ and is less than $2ulp$. For $x \in [2^{-11}, 8)$ an unsigned fixed-point format having width 37 and fraction 34 allows representing the SP FP values in this range without any accuracy loss.

We apply the following range reduction:

$$\tanh(a+b+c) = \frac{\tanh(a) + \frac{\tanh(b)+\tanh(c)}{1+\tanh(b)\tanh(c)}}{1 + \tanh(a)\frac{\tanh(b)+\tanh(c)}{1+\tanh(b)\tanh(c)}} \qquad (11)$$

We choose a, b, c which allow us to simplify Eq. 11:



The weights of b and c allows us to approximate the denominator $1+\tanh(b)\tanh(c)$ by 1. The largest value for the product $\tanh(b)\tanh(c)$ occurs for the all ones mantissa aligned on b and c; in this case the product has weight less than 2^{-24}. Additionally, the range of $c \in [2^{-17}-2^{-34}, 0]$ allows for approximating $\tanh(c) \approx c$. Consequently Eq. 11 becomes:

$$\tanh(a+b+c) = \frac{\tanh(a)+\tanh(b)+c}{1+\tanh(a)(\tanh(b)+c)} \qquad (12)$$

This equation is to be implemented as a product between the numerator and the reciprocal of the denominator. The size of the reciprocal can be reduced by observing that the ranges of the multiplication terms: $\tanh(b)+c \in [0, 2^{-7})$ and $\tanh(a) \in [0,1]$ leads to the product belonging to $[0, 2^{-7})$. The reciprocal is therefore computed for $1/1+y$ with $y \in [0, 2^{-7})$ and a 26-bit accurate output.

The 37 bit fixed-point representation of the input is obtained using a barrel shifter with a max-shift value of 15. Beyond this shift value the function returns x. The values $\tanh(a)$ and $\tanh(b)$ are obtained by tabulation; $\tanh(a)$ is stored on 34 bits (all fractional) and $\tanh(b)$ is stored on 27 bits, 34-bit fraction.

The numerator is obtained by summing the 3 fixed-point values; the format of the sum is 35 bits, 34 fractional. For the denominator we only need to focus on the right-hand term $(\tanh(a)(\tanh(b)+c))$ as only this value will be used by the custom reciprocal unit. Since we will only use the top 16 bits of the product, we truncate the inputs in order to reduce multiplier count. We select the top 18 bits of the tabulated value $\tanh(a)$ together with the top 18 bits of the sum $\tanh(b)+c$. Our custom reciprocal unit will input 16 bits and will output 26 bits. We use a piecewise polynomial approximation (PPA) implementation that requires a degree 1 polynomial; the number of subintervals is 64 and the multiplication size is 11×15.

A coarse-grain normalization of the numerator is performed before the final multiplication. The maximum left shift is 12 positions. The multiplication fits well in one 27×27 multiplier. A final fine-grain alignment (by at most one binary position) is required for packing the FP result. The full architecture and the fixed-point alignments are shown in Figure 3.

B. Hard FP DSP Blocks

A new set of implementations are possible on target devices that contain HFP DSP Block support. We now describe HP and SP architectures based on piecewise floating-point polynomial approximations (FP PPA). We obtain our polynomials using the Sollya tool [15] by means of the

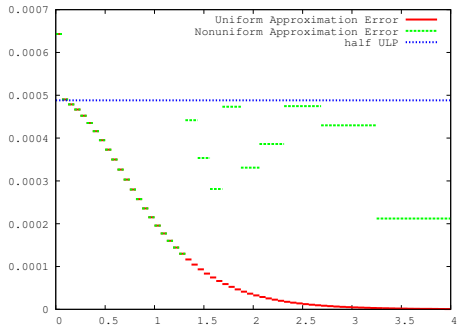


Figure 4. HP $\tanh(x)$ approximation error for uniform and non-uniform segmentation schemes

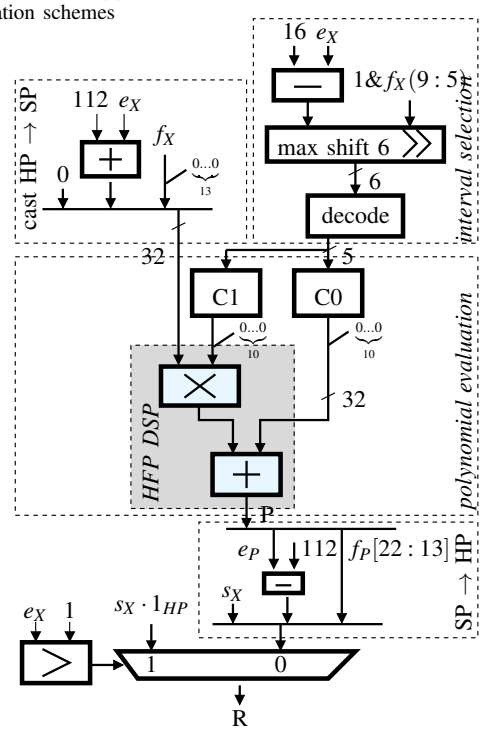


Figure 5. HP $\tanh(x)$ architecture using Hard FP DSP Blocks

`fpminimax` procedure [16]. The approximation error is evaluated on each interval using the `infnorm` function [17]. The evaluation error is computed using Gappa [18] on each subinterval.

1) Half-Precision: We implement the function on the interval $[0, \approx 3.81]$. For values larger than ≈ 3.81 we return 1. For values smaller than 2^{-5} we return the input. We reconstruct the results for the negative input range by appending the input sign to the result.

The input interval, extended to $[0,4)$ is split into 64 subintervals. On each subinterval a degree 1 polynomial having single-precision coefficients is used to approximate *tanh*. The approximation error is plotted using solid lines in Figure 4. It can be observed that the approximation error is significantly better as we approach the right of the approximation interval. We can reduce the number of stored coefficients by using a non-uniform segmentation scheme; as

long as the complexity associated with the interval decoding is low, this scheme can reduce ALM count. In Figure 4 the approximation error and the interval sizes for a non-uniform segmentation scheme are plotted. It can be observed that the number of subintervals used on the right side of the input interval is reduced. Overall, the number of subintervals is decreased to 30; the cost of decoding the subinterval index is a 5-bit LUT6 table lookup.

Since the HFP Block allows for single-precision multiply-add, we store the polynomial coefficients in SP format. Only the upper 13 mantissa bits are explicitly stored, with the remainder zero padded the at the output of the tables. This nearly halves the ALM table count, but the impact on the approximation accuracy is minor.

The architecture is depicted in Figure 5. A small barrel shifter is used to create a 6-bit fixed-point value that identifies 64-subintervals. The *decode* table maps the 6-bit fixed-point to a 5-bit fixed-point value that identifies the table address storing the coefficients for the non-uniform intervals. The HP input is cast to SP by adjusting its exponent bias offset (adding 127-15) and padding the fraction with 13 zeros. The coefficient tables only output a 13-bit fraction. Consequently, at the output of these tables the coefficients have their fractions zero extended to the right before feeding these into the polynomial evaluator. The evaluation is then mapped directly on the Hard FP DSP Block Mult-Add pair. The output of the evaluation is in SP and needs to be cast back to HP. This is done by adjusting the exponent (subtracting the bias offset 112) and truncating the fraction to 10 bits. The accuracy can be improved at this stage by rounding the SP mantissa to the nearest HP mantissa, which may involve an exponent update. The input sign is then added to this half-precision output value.

Finally, a multiplexer selects between the implementation branches. For exponent values larger or equal to 2 we return $s_x \times 1.0$. The branch that returns x if input $< 2^{-5}$ is fused with the polynomial approximation; one extra set of polynomials (C1=1, C0=0) is stored for the case the 6-bit input is all zeros. The exact details are omitted from Figure 5 for clarity.

2) Single-Precision: The SP architecture is centered around a degree-5 odd FP PPA. For inputs larger than 8 we approximate the function with the value 1. For inputs smaller than 2^{-12} we return the input. The degree-5 polynomial is evaluated as follows:

$$P(x) = x(c_1 + x^2(c_3 + x^2 c_5))$$

For a sufficiently accurate approximation, the interval $[0,8)$ is split in 512 subintervals. The approximation error is depicted in Figure 6 for segmentations with both 512 and 256 subintervals. The maximum approximation error is between $1/4ulp$ and $1/8ulp$ for 512 subintervals. For 256 subintervals the maximum approximation error is approximatively $1.5ulp$.

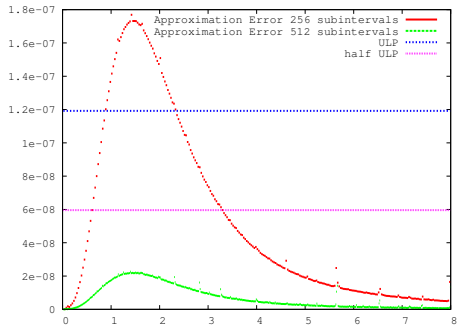


Figure 6. Single-precision $\tanh(x)$ approximation error for uniform segmentation scheme with 512 and 256 subintervals on the input interval $[0,8)$

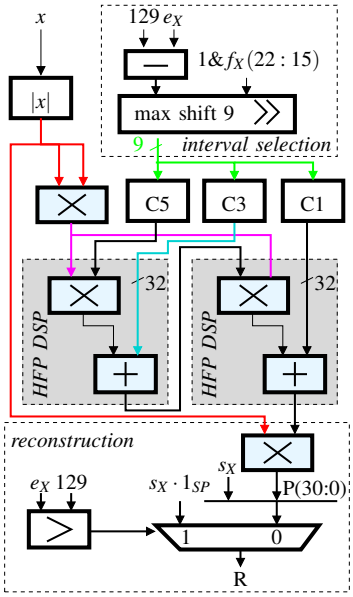


Figure 7. Single-precision $\tanh(x)$ architecture using Hard FP DSP Blocks

The interval index is obtained by converting the input to a 9-bit fixed-point value having 3 integer and 6 fractional bits; this is accomplished by feeding the top 9 input mantissa bits to a small barrel shifter with the shift value set to $2 + bias - e_X$. The 9-bit address is used to fetch the 3 polynomial coefficients that are each stored into a memory block (the M20K block allows storing 512 coefficients of up to 40 bits).

The single-precision architecture is depicted in Figure 7. The coefficients map directly to M20K blocks; the polynomial evaluation maps onto 4 DSP blocks: one used for the initial squaring operation, two in mult-add mode, and the final one in stand-alone multiplier mode.

V. Sigmoid

The sigmoid function is defined by the identity of Eq. 13.

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{13}$$

Similar to tanh, as x increases $x > x_{\max}$ is a good approximation for sigmoid is 1; as x decreases, say $x < x_{\min}$, a good

approximation for the returned result is 0. Consequently, the compute branch for the sigmoid will need to return values for the range $[x_{\min}, x_{\max}]$.

In a typical FP implementation $|x_{\min}| \neq |x_{\max}|$ since the density of FP values close to 0 allows for more output values to be represented before the output saturates to 0. However, the architectures proposed in this work will use a symmetric compute range $[-x_{\max}, x_{\max}]$ and we therefore approximate the sigmoid with 0 in the range $[x_{\min}, -x_{\max})$.

The bound $x_{\max}$ is the first value for which the function returns 1; the denominator $1+e^{-x}$ will therefore need to be 1, or in other words e^{-x} will need to be shifted out when being added to 1, which leads to the inequality $e^{-x} < 2^{-wF-1}$. Therefore, for $x > (wF+1)log(2)$ the sigmoid function returns 1 with an accuracy of $1/2ulp$. The bound may be relaxed so to aim for an accuracy or $1ulp$ around the value $x_{\max}$. The HP and SP bounds for $x_{\max}$ are listed below.

precision	$< 1/2ulp$	$< 1ulp$
half (wF=10)	7.62	6.93
single (wF=23)	16.63	15.94

We can focus the implementation to the negative range $(-x_{\min}, 0]$ and reconstruct the result for the positive range using the following identity:

$$f(x) = 1 - f(-x) \tag{14}$$

Selecting the reconstruction starting from the negative range is crucial for maintaining accuracy. Conversely, if reconstruction is done starting from the positive range, the accuracy on the negative (reconstructed) range varies from wF bits for x close to 0 down to 1 bit for x close to $-x_{\max}$. Alternatively, one can evaluate the positive range in parallel for reduced latency.

A. Half-precision

A FP PPA can be used for the negative range $(-8, 0]$. With 64 subintervals and a degree-1 polynomial the approximation error is bounded by $1.25ulp$. The reconstruction of the result for the positive range is accomplished using one floating-point subtracter. The architecture is depicted in Figure 8(a) and corresponds to the (sub) architecture in Table I.

The latency of this first proposed architecture is dominated by the chaining of the 2 FP DSPs. This latency may be reduced by evaluating in parallel both the positive and negative ranges. The total number of DSPs is still 2, as a multiply-add costs as much as the subtracter. This architecture corresponds to the *eval both unif* entry in Table I. The latency is reduced from 8 cycles down to 5 cycles.

The two evaluation datapaths (for the positive and negative ranges) can be fused in order to reduce the number of DSPs down to 1. Additionally, a non-uniform segmentation for the positive range allows reducing the number of subintervals used to 14 with an approximation error of less

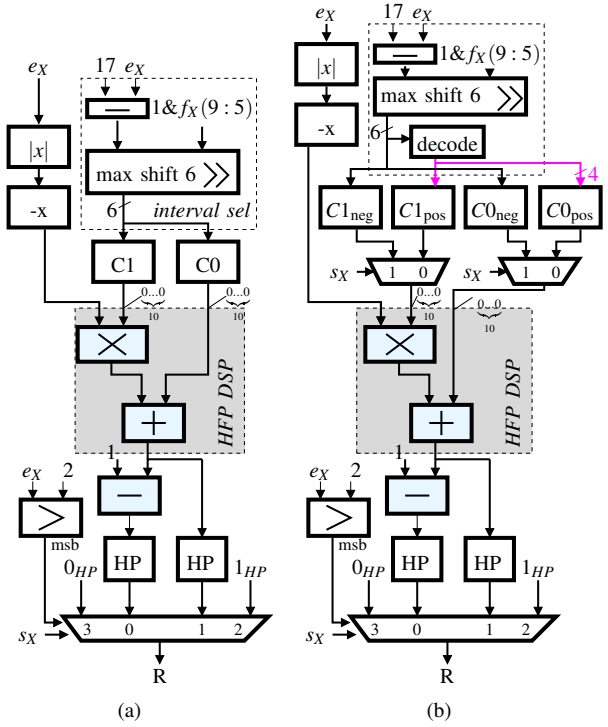


Figure 8. HP sigmoid(x) architecture using Hard FP DSP Blocks

than $1ulp$. The decode table outputs the 4 bits required for indexing the positive range tables. Multiplexers driven by the sign bit make the selection between the negative and positive-range coefficients. This architecture is depicted in Figure 8(b) and corresponds to line *eval fused non-unif* in Table I.

B. Single-Precision

The intuition is to use the same type of PPA as for the HP implementation. For single-precision the interval of interest is $(-16, 0]$ as the function values for the positive range get reconstructed starting from these (Eq. 14). Let P_i be the corresponding polynomial that approximates the function sigmoid(x) for $x \in [L_i, R_i]$. A good bound on the approximation error can be obtained using either Minimax or Taylor degree 3 polynomials and 256 subintervals. Unfortunately, evaluating the P_i polynomials using SP arithmetic results in very low accuracies.

The wide discrepancy between the approximation and evaluation error is plotted in Figure 9 for the degree 3 Taylor approximation (plotted only on the interval $(-8, 0]$); here the y axis plots $-\log_2(\max(E_{\text{rel}}))$ for each subinterval and corresponds intuitively to the accuracy in binary digits of the plotted approximations. The solid red line shows the approximation error which is translates into approximately $1/4ulp$ maximum error. The dashed green lines (the cloud on the bottom right of the figure) show the sum of approximation and evaluation error for each these Taylor polynomials; this is much worse than the approximation error alone and

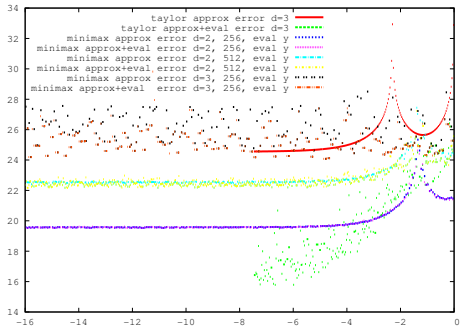


Figure 9. SP PPA approximation and evaluation errors for sigmoid(x)

translates into a worst-case accuracy of roughly 16 bits on the depicted range.

In order to improve the overall accuracy we modify the function to be evaluated by each polynomial. For each subinterval $[L_i, R_i]$ the function sigmoid(x) is approximated by a polynomial $P_i(y)$ where $y = x - L_i$, and $y \in [0, R_i - L_i]$. The argument change on the input of P_i results in a very tight evaluation error. For obtaining a result within 4 ULPs, an degree 2 Minimax approximation using 512 subintervals is sufficient.

The architecture requires a FP subtracter for computing the new input argument for the polynomials $P_i(y)$. The right operand of the subtraction L_i is the left bound for each approximation subinterval, stored as a float. The values L_i are tabulated and stored similarly to the coefficients C_0, C_1, C_2. This design would require four M20K blocks, one for each: C_0, C_1, C_2, L_i. This count can be reduced to three blocks by first observing that L_i has 15 trailing zeros, and 1 leading one; the 32-bit float can be created by padding these to the tabulated 16-bits for L_i. Also, the C_1, C_2 and L_i tables can be fused; the total width is (32+32+16)=80 which is exactly the width of two M20Ks in 512x40-bit mode. The full architecture is depicted in Figure 10.

VI. Results

Table I presents the synthesis results for our proposed architectures. We have selected the Stratix V family for the integer arithmetic only target, and the Arria 10 devices for the HFP case. The synthesis results have been obtained using Quartus Prime Standard Edition 17.1 for the Stratix V devices, and Quartus Prime Pro Edition 17.1 for Arria 10 devices. Both devices have been selected with the fastest core speedgrade (-1). We have pipelined our presented architectures for expected maximum device frequencies; a smaller area and a lower latency can be obtained at the expense of reduced performance if the system application warrants it.

First, we have compared our implementations against operator-assembly-based floating-point architectures. To our knowledge, the most efficient cores targeting Intel FPGAs

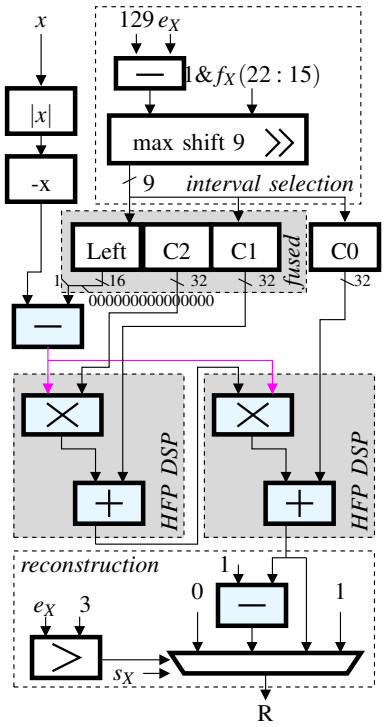


Figure 10. SP sigmoid(x) architecture using Hard FP DSP Blocks

are available from the DSP Builder Advanced toolbox [19]. We have therefore used this Simulink-based flow to architect the operator assembly examples for the two functions. As expected, our new architectures are significantly more efficient.

We have also compared our work with the few cores found in the literature. [12] presents results for a 64-bit fixed-point with a 42-bit fraction. For *tanh* their reported maximum approximation error (absolute) is 1.695e-7 (or $\approx 2^{-23}$) which makes it comparable to our single-precision implementation. Their core not only uses more logic but also consumes significantly more DSP resources. The same conclusion holds for the sigmoid function but their reported maximum absolute approximation error is now improved to 9.97e-11 (or $\approx 2^{-34}$). This is expected since their fraction length is 42-bit, meaning that in the worst-case the 8 trailing bits of the result will be invalid.

In the case of sigmoid [13] describes an multiplier-based architecture, for single-precision. The reported results show a much higher LUT and DSP count than in our presented implementations.

VII. Conclusions

By analyzing the performance of RNNs, we have shown the need for efficient, high performance - but at the same time, low latency - activation functions. We have presented a number of new architectures for the *tanh* and sigmoid functions. On FPGA devices with integer arithmetic only, the presented architectures for *tanh* significantly lower resource

Table I
RESOURCE UTILIZATION AND PERFORMANCE FOR THE PROPOSED ARCHITECTURES

Function	Precision	Arch.	Target	Lat.@ Freq.	Resources
tanh	HP	Prop. No Hard FP DSP	Stratix V	9 @ 549MHz	97 ALMs, 1 DSP, 0 M20Ks
		Assembly $e^x, \div, +$		32 @ 504MHz	559 ALMs, 2 DSP, 0 M20Ks
		Prop. Hard FP DSP	Arria 10	5 @ 483MHz	54 ALMs, 1 DSP, 0 M20Ks
	SP	Prop. NoHardDSP	Stratix V	17 @ 430MHz	348 ALMs, 2 DSP, 4 M20Ks
		Assembly $e^x, \div, +$		47 @ 451MHz	770 ALMs, 6 DSP, 12 M20Ks
		Hard FP DSP	Arria 10	18 @ 483MHz	177 ALMs, 4 DSP, 3 M20Ks
	64-bit fxp, 42-bit fraction	[12]	Virtex 5	? @ ? MHz	1687 LUTs, 34 DSP
sigmoid	HP	Hard FP DSP (sub)	Arria 10	8 @ 483MHz	80 ALMs, 2 DSP, 0 M20Ks
		Hard FP DSP (eval both unif)		5 @ 483MHz	78 ALMs, 2 DSP, 0 M20Ks
		Hard FP DSP (eval fused non-unif)		5 @ 483MHz	93 ALMs, 1 DSP, 0 M20Ks
	SP	Hard FP DSP	Arria 10	22 @ 483MHz	222 ALMs, 4 DSP, 3 M20Ks
		[13]	Zed7020	? @ ≈18MHz	2018 LUTs, 21 DSP, 1 RAM
	64-bit fxp, 42-bit fraction	[12]	Virtex 5	? @ ? MHz	1388 LUTs, 22 DSP

utilization and latency compared to operator-assembly-based architectures. On devices with embedded HFP, we have shown further optimizations on resources and latency.

Numeric accuracy is important. While not performing a formal error analysis, we have verified the approximation errors of the FPMinimax polynomials generated by the Sollya tool with `infnorm` functions. We have also checked the evaluation error using the Gappa tool. This was particularly important as we have observed a very wide gap between the two errors for the sigmoid approximation.

REFERENCES

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

[2] F. de Dinechin and B. Pasca, "Floating-point exponential functions for DSP-enabled FPGAs," in *Field-Programmable Technologies*. IEEE, 2010.

[3] M. Langhammer and B. Pasca, "Single precision logarithm and exponential architectures for Hard Floating-Point enabled FPGAs," *IEEE Transactions on Computers*, vol. 66, no. 12, pp. 2031–2043, Dec 2017.

[4] B. Pasca, "Correctly rounded floating-point division for DSP-enabled FPGAs," in *22th International Conference on Field Programmable Logic and Applications (FPL'12)*. Oslo, Norway: IEEE, Aug. 2012.

[5] *Arria10 Device Overview*, 2014, http://www.altera.com/literature/hb/arria-10/a10_overview.pdf.

[6] *Stratix10 FPGA and SoCs*, 2015, https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html.

[7] M. Langhammer and B. Pasca, "Faithful single-precision floating-point tangent for FPGAs," in *Proceedings of the 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2013, pp. 39–42.

[8] "Understanding LSTM networks," http://colah.github.io/posts/2015-08-Understanding-LSTMs/, accessed:18-05-28.

[9] M. Langhammer and B. Pasca, "Floating-point DSP block architecture for FPGAs," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 117–125. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689071

[10] M. Papadonikolakis and C. S. Bouganis, "A scalable FPGA architecture for non-linear SVM training," in *2008 International Conference on Field-Programmable Technology*, Dec 2008, pp. 337–340.

[11] Z. Hajduk, "High accuracy FPGA activation function implementation for neural networks," *Neurocomputing*, vol. 247, pp. 59 – 61, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231217305593

[12] V. Tiwari and N. Khare, "Hardware implementation of neural network with sigmoidal activation functions using CORDIC," *Microprocessors and Microsystems*, vol. 39, no. 6, pp. 373 – 381, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933115000642

[13] L. Zhang, "Implementation of fixed-point neuron models with threshold, ramp and sigmoid activation functions," *IOP Conference Series: Materials Science and Engineering*, vol. 224, no. 1, p. 012054, 2017. [Online]. Available: http://stacks.iop.org/1757-899X/224/i=1/a=012054

[14] J. Harrison, "A machine-checked theory of floating point arithmetic," in *Theorem Proving in Higher Order Logics*, 1999, pp. 113–130.

[15] S. Chevillard, C. Lauter, and M. Joldes, "Users manual for the Sollya tool, Release 2.9," http://sollya.gforge.inria.fr/, February 2011.

[16] N. Brisebarre and S. Chevillard, "Efficient polynomial l^∞-approximations," in *18th IEEE Symposium on Computer Arithmetic (ARITH 18)*. Los Alamitos, CA: IEEE Computer Society, June 2007, pp. 169–176.

[17] S. Chevillard, J. Harrison, M. Joldes, and C. Lauter, "Efficient and accurate computation of upper bounds of approximation errors," *Theoretical Computer Science*, vol. 412, no. 16, pp. 1523 – 1543, 2011.

[18] F. de Dinechin, C. Lauter, and G. Melquiond, "Certifying the floating-point implementation of an elementary function using Gappa," *IEEE Transactions on Computers*, pp. 1–14, 2010.

[19] "DSP Builder Advanced Blockset," 2017, https://www.altera.com/products/design-software/model---simulation/dsp-builder/overview.html.