COMPACT: Co-processor for Multi-mode Precision-adjustable Non-linear Activation Functions

Wenhui Ou[†‡], Zhuoyu Wu[†], Zheng Wang[†*], Chao Chen[†*], Yongkui Yang[†]
[†]Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China
[‡]School of Mechanical Science and Engineering, Huazhong University of Science and Technology, Wuhan, China
[*]Zheng Wang and Chao Chen are the corresponding authors, Email: (zheng.wang, chao.chen)@siat.ac.cn

***Abstract*—Non-linear activation functions imitating neuron behaviors are ubiquitous in machine learning algorithms for time series signals while also demonstrating significant gain in precision for conventional vision-based deep learning networks. State-of-the-art implementation of such functions on GPU-like devices incurs a large physical cost, whereas edge devices adopt either linear interpolation or simplified linear functions leading to degraded precision. In this work, we design COMPACT, a co-processor with adjustable precision for multiple non-linear activation functions including but not limited to exponent, sigmoid, tangent, logarithm, and mish. Benchmarking with state-of-the-arts, COMPACT achieves a 26% reduction in the absolute error on a 1.6x widen approximation range taking advantage of the triple decomposition technique inspired by Hajduk's formula of Padé approximation. A SIMD-ISA-based vector co-processor has been implemented on FPGA which leads to a 30% reduction in execution latency but the area overhead nearly remains the same with related designs. Furthermore, COMPACT is adjustable to 46% latency improvement when the maximum absolute error is tolerant to the order of 1E-3.**

***Index Terms*—Approximate computing, non-linear functions, architecture exploration**

I. Introduction

The previous decade has witnessed the huge success of artificial neural networks (ANNs), which have been increasingly deployed on data centers, mobile edges, and terminal devices. Out of numerous evaluation aspects, inference precision, performance, and energy efficiency are primary considerations for executing ANNs which motivate the design and fabrication of customized accelerators. Systolic-Array-based architectures such as Tensor Processing Unit (TPU) [1] are specialized in executing ***linear*** operations such as convolution and matrix multiplication with massive parallelism. Attribute to the advancement of quantization techniques [2] which enables *fixed-point (FXP)* inference with tiny precision degradation, the implementation of processing elements in TPU is extremely compact to house tens of thousands of multipliers on the same die.

Albeit excellent in performance, TPU has weak support of ***non-linear*** activation functions (NAFs). Consequently, simplified linear activation functions e.g. ReLU/leaky-ReLU are widely deployed instead of neuroscience-inspired NAFs. Although such simplification is tolerable for early vision-based ANNs, recent ANNs tend to achieve high precision with the aid of complex NAFs. For instance, Yolo-v4 [3] and onward versions [4] are built with Mish functions, while tanhExp significantly accelerates the training process of lightweight neural network [5]. Sigmoid and Tanh steadily resides in recurrent networks for time series signal processing [6]. Such trend implies that sophisticated NAFs have become the dominant factor to achieve high precision.

Most current inference accelerators lack architectural support for NAFs. Piecewise linear (PWL) approximation techniques are employed in [7] [8] for specific NAFs. In contrast, Graphics Processing Unit (GPU) such as Fermi equips Special Function Units (SFU) to support single-precision NAF [9], which inspires us to augment the baseline architecture with a dedicated co-processor for NAF. SIMD-based instruction sets can be utilized with both vector and array-style data-parallel processors to increase the throughput of execution.

The major design challenge for the co-processor lies in the programmability to support broad variants of NAFs as well as the flexibility to adjust the precision and fitting range. It is based on the observation that the requirement of the numerical gap between theoretical and approximate values, known as absolute approximation error (AAE) [10], varies among application scenarios. For instance, an AAE of PWL approximation around 1E-4 is sufficient for Yolo post-processing [7] where neural network training demands an AAE of less than 1E-6 [11]. Furthermore, the range of inputs with AAE less than a certain value, known as approximation range (AR) [10], also varies but is crucial to application-level accuracy. Adjustable AAE and AR can be used to trade-off with latency, for instance, low latency upon a relaxed precision requirement.

In this work, we present COMPACT, a co-processor with adjustable precision for multiple NAFs. As illustrated in Fig. 1, COMPACT works synergetically with an exploration framework, where applications are first analyzed to extract non-linear operations. The workbench of COMPACT takes NAFs and user constraints as inputs and generates co-processor RTL and associated SIMD instructions to achieve high flexibility. The user can further integrate COMPACT with conventional TPUs which execute linear operations, therefore the whole application can be deployed with both high performance and sufficient precision.

The core of the COMPACT workbench lies in its approximation engine, where we derive mainstream NAFs from exponent (*exp*) and natural logarithm (*ln*) kernels. AAE and AR of functions are conveniently adjusted by tuning the expansion forms of both kernels, while a widened AR can be achieved with a triple decomposition of *exp* kernel inspired by Hajduk's decomposition [10] of Padé approximation [12]. COMPACT is evaluated through executing generated SIMD instructions on an FPGA prototype of the co-processor, which simultaneously achieves less AAE, wider AR, and lower latency than state-of-the-art implementations.

The rest of the work is organized as follows. Section II provides background information on approximation of NAFs. Section III details the approaches in approximation engine and

978-3-9819263-7-8/DATE23/© 2023 EDAA

the exploration of design trade-offs. Section IV illustrates the proposed vector processor. Section V presents the experiments and benchmarking results. Section VI concludes the work.

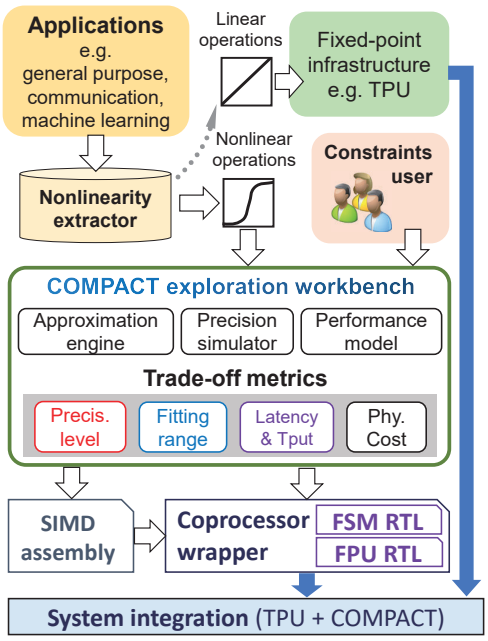


Fig. 1: Framework of COMPACT for co-processor exploration

II. Background

NAFs such as sigmoid, hyperbolic tangent (tanh), tanhExp and mish–as shown in equation (1), (2), (3) and (4)–are widely adopted in state-of-the-art applications and play an essential role in enhancing the algorithmic precision.

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

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

$$\text{tanhExp}(x) = x \times \tanh(e^x) \tag{3}$$

$$\text{mish}(x) = x \times \tanh(ln(1+e^x)) \tag{4}$$

PWL approximation is an efficient way to simplify the non-linear behaviour of a function and it is widely adopted for NAFs [13] [14]. However, its memory volume requirement is costly when it comes to highly-precise approximation, as the number of Look-Up Tables (LUTs) increases with accuracy. For the lack of flexibility for different functions, the storage overhead will further increase when implementing various of NAFs.

To amortize this penalty, some studies have narrowed their focus to the composition of these NAFs. Gomar et al. [15] proposed an exponential approximation-based method to calculate sigmoid/tanh functions with relatively lower accuracy. The implementation of *exp* operation is converted into a base 2 equation so that it can be performed by addition and shift operations using FXP units. Pan et al. [16] proposed FXP hardware implementation of the sigmoid function based on the Newton-Raphson method. As the iterative equation consists of sigmoid function and *exp* operation, they used four-segment PWL to approximate sigmoid and computed *exp* terms by Taylor series. In [10], Hajduk applied LUTs with either the Mclaurin series (i.e. Taylor series at the point zero) or Padé polynomials to approximate *exp* operation in a split domain for high precision realization. However, the argument domain after reduction is still large, which is not sufficient for a small enough degree of polynomials while resulting in long latency.

III. Approximation Engine

For a better performance in the high-precise execution of NAFs, we have developed an approximation engine based on Hajduk's *exp* approximation formula. On the one hand, the *floating-point (FP)* representation is exploited to make the argument decomposition more efficient. On the other hand, the split domain is further reduced for a lower degree of the polynomial, which will significantly improve execution time at a little cost. In addition, we noticed that the various degree of the approximate equation can offer different levels of speed-accuracy trade-off via the decomposition method. To this end, different polynomials are implemented in a combinational way to reduce the redundant effort from the changing demand of precision in inference [17]. In addition to *exp* operations, mish's operation includes *ln* operation, which faces the same challenge as *exp*. It should be noted that *ln* can also be computed in a similar way to *exp* operations disassembled, which has rarely been mentioned in related work.

A. exp triple decomposition

For rational function approximation, the value of e^x can be computed by the Taylor series or Padé polynomial using equation (5) and (6), respectively.

$$e_n(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} \tag{5}$$

$$R(x) = \frac{\sum_{j=0}^{m} a_j x^j}{1+\sum_{k=1}^{n} b_k x^k} = \frac{a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m}{1 + b_1 x + b_2 x^2 + \cdots + b_n x^n} \tag{6}$$

Unfortunately, the approximate function is valid for a narrow interval nearby zero and the width of the interval is affected by the degree of function. In this design, we use equation (7) for a widened AR.

$$e^x = e^p \times e^{d=t+f} = e^p \times e^t \times e^f \tag{7}$$

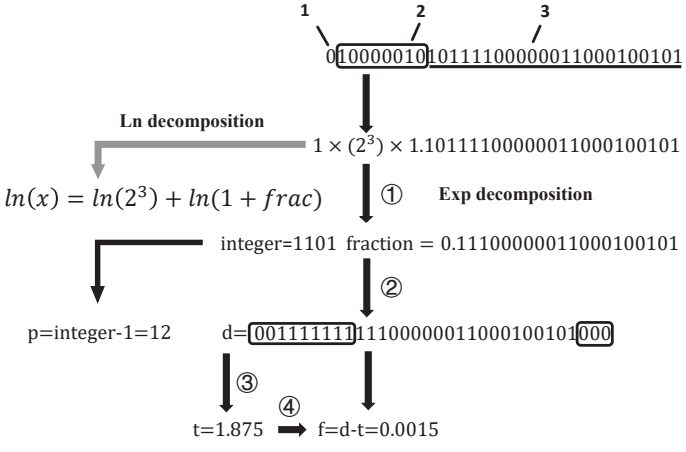
where $p = \lfloor x \rfloor - 1$, $d = x - p$. With further decomposition, d can be divided as t and f. As the *unit-in-last-place (ulp)* of t is 2^{-3}, f can be limited to $(-0.125, 0.125)$, and e^f is satisfied for equation (5) and (6). For the other items, e^p and e^t can be computed by storing the numerical value in the LUTs ahead of time. Fig. 2 displays the overall decomposition with an example (note, the input x=13.8765 expressed as FP32):

- According to the binary FP representation of input x, eight exponent bits are acquired. Then subtract the constant offset (i.e. 127), the binary representation of x can be expressed as $2^3 \times (1 + frac)$. And the calculation of the equation only needs three left shift operations based on FXP arithmetic. Therefore, the integer and fraction parts are acquired in FXP format.
- For the preparation of the next step, the fraction needs to be extended into FP32. As shown in Fig. 2, where the exponent part is set as zero (including the constant offset), and also padding zeros behind the fraction part based on the round to zero mode. Therefore, the fraction part expressed as d is transformed from FXP to FP32. As the value of the

fraction part adds one passively after transformation, the integer needs to subtract one correspondingly and we use p to denote that.

- As t is used for reducing the interval of f, the bit-width of t (represented in n), depends on the required accuracy and hardware cost. For comprehensive consideration, we set $n = 4$, where including one integer bit and the rest bits refer to the fraction part. With the relative position of integer bits between d and t, the value of t is received, i.e. $t = 0b1111 = 1.875$, as shown in Fig. 2. With a similar transformation, t can be extended into FP32 format, and f is acquired by d - t based on FP subtract.

Consequently, we have divided x into three items (i.e $x = p + t + f$) corresponding to the $e^x = e^p \times e^t \times e^f$. In addition, the overall decomposition flow only includes an FP operation, therefore it can be performed in several cycles. Note that e^{-16} or e^{16} is directly output for $|x| \geq 16$, based on the sign bit. And for $|x| < 2$, e^p is omitted.



1	*sign* = 0
2	*exponent* = 130+(-127)= 3
3	*fraction(frac)*
①	*left shift operation*
②	*transformation operation*
③	*extracted operation*
④	*transformation operation*

Fig. 2: Decomposition flow in the approximation engine

B. ln decomposition

The *ln* operation can be further divided into small instances by equation (8).

$$ln(x) = ln(2^a \times b) = ln(2^a) + ln(b) \tag{8}$$

The value of $ln(2^a)$ can be derived from LUTs, and $ln(b)$ is approximated by Padé or Taylor series. Similarly, the conversion from x to $2^a \times b$ is based on the binary representation of FP as discussed in subsection III-A. Initially, the exponent part of the input is extracted and minus the offset value. In the next step, the rest bits (the fraction part of the binary representation of x), are converted into FP32 format with the transformation operation mentioned before. As shown in Fig. 2, the expression can be rewritten as:

$$ln(x = 13.8765) = ln(2^3) + ln(1 + frac) \tag{9}$$

In the process, b is limited to [1,2) and the range of a is set as [-16,16] (note, the constant value (i.e. $ln(2^{16})$ or $ln(2^{-16})$) is output for overflow). And the overall decomposition can be completed in one cycle.

C. Analysis of approximation error

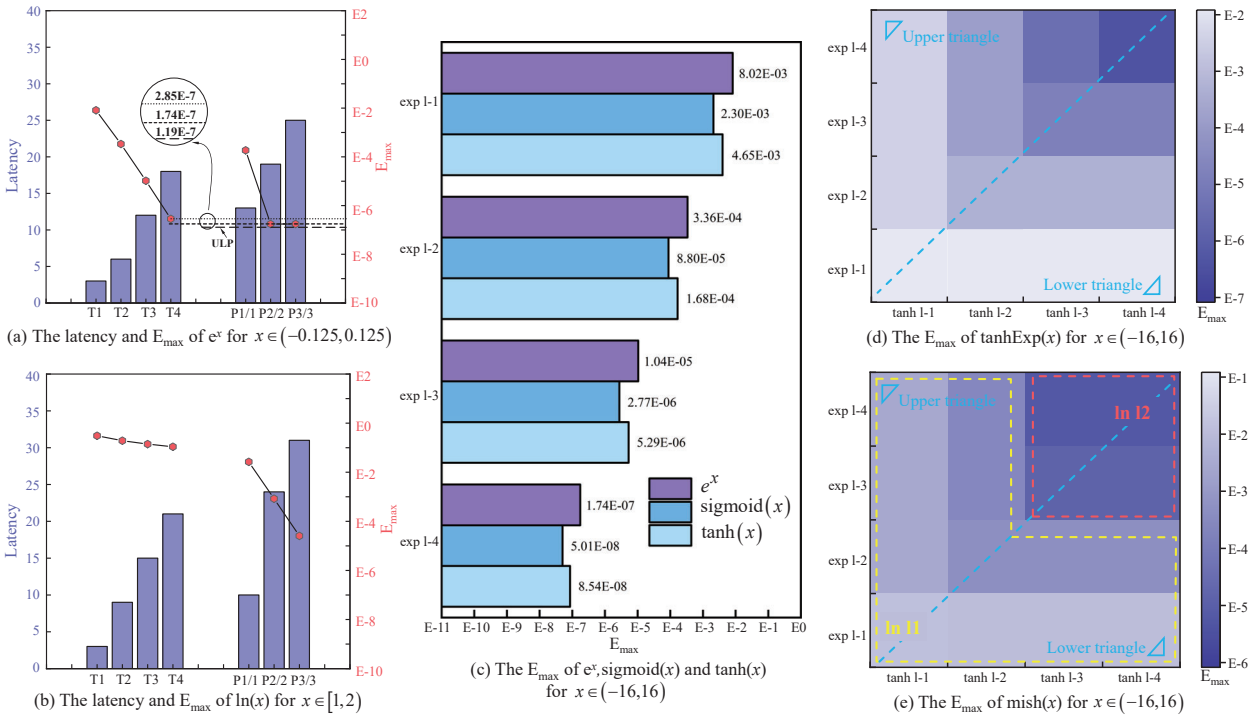
To match the needs of programmers as possible, it is necessary to make effort in the diversity of the precision choices. In this regard, we have carried out a set of simulation experiments in Python to analyze the AAE of equation (7) and (8) with different approximate functions. For better cost-effectiveness, the propagation of the error in different NAFs will be further analyzed in the following.

$$AAE(x_i) = |\tilde{y}(x_i) - y(x_i)| \tag{10}$$

$$E_{max} = \underset{i=1,\cdots,N}{max}(AAE(x_i)) \tag{11}$$

We use equation (10) to define the AAE for value x_i, where $\tilde{y}(x_i)$ is the approximate value and $y(x_i)$ is the baseline value obtained using Python. Equation (11) defines the maximum absolute approximation error (MAAE) as E_{max}, where N is the number of sampling points. Fig. 3 displays the MAAE analysis and they are summarized as follows.

- Fig. 3 (a)(b) compares the E_{max} of *exp* and *ln* operations, respectively. For *exp* on the small domain, Padé$(2,2)$ is enough. Between the order of E-2 and E-6, the Taylor series shows better efficiency. But for *ln* with the larger AR, Padé provides a balanced approximation instead of Taylor leading to the error exploding. Out of different contributions from Padé and Taylor, we wisely adopted both to provide diversity for mixed accuracy strategies. In conclusion, Padé$(2,2)$, Taylor(3), Taylor(2), Taylor(1) are used as e^f approximation formulas for 4-level precision, and Padé$(3,3)$, Padé$(2,2)$ are used as $ln(b)$ approximation formulas for 2-level precision.
- For sigmoid and tanh that includes *exp*, MAAE mainly depends on the precision of *exp* (Fig. 3 (c)). Even in the lower approximation level, the MAAE is further reduced. For instance, the E_{max} of *exp* is 3.36 E-4 with precision level 2, but that of the sigmoid is 8.80 E-5.
- For the remaining NAFs with the more sophisticated component, i.e. tanhExp (Fig. 3 (d)) and mish (Fig. 3 (e)), the errors are slightly magnified. It is mostly attributed to the approximate operations in chain resulting in error accumulation. In addition, our profiling has shown that applying higher precision to the specific operation may achieve better accuracy for NAFs. For instance, the upper triangular sections in chart have overall lower errors than the lower triangular section, albeit consistent in execution time.



(a) The latency and E_{max} of e^x for $x \in (-0.125, 0.125)$

(b) The latency and E_{max} of $\ln(x)$ for $x \in [1, 2)$

(c) The E_{max} of e^x, sigmoid(x) and tanh(x) for $x \in (-16, 16)$

(d) The E_{max} of tanhExp(x) for $x \in (-16, 16)$

(e) The E_{max} of mish(x) for $x \in (-16, 16)$

Fig. 3: (a), (b) The latency and E_{max} of e^x ($x \in (-0.125, 0.125)$) and $ln(x)$ ($x \in [1, 2)$), with different approximation schemes; (c) The E_{max} of e^x, sigmoid(x) and tanh(x) for $x \in (-16, 16)$ with different exp precision levels; (d), (e) respectively corresponding to the E_{max} of tanhExp(x) and mish(x) for $x \in (-16, 16)$, calculated with different precision levels of the component. (Note, Taylor(n) expressed as Tn and Padé(m,n) expressed as Pm/n. The precision level n is expressed as l-n.)

IV. Architecture

The proposed System-on-Chip (SoC) consists of the COMPACT co-processor, a TPU engine, a DRAM interface, a DMA controller, and peripherals as illustrated in Fig. 4a. TPU is mostly equipped with array-style processing elements to work in a high throughput mode. The weakness in the throughput of the coprocessor may impact the execution of TPU. COMPACT encompasses four spatial FPUs and each of them can perform temporal vector operations, which is contribute to the proper balance in throughput for most cases. For the elementary arithmetic support of COMPACT, an FP adder, multiplier, subtractor and divider are implemented on each FPU, as shown in Fig. 4b. Although FP operations always refer to long latency in the generic programming, it can be overlapped by the long vector operations, for example, the FP division in a pipelined manner can output per cycle.

Fig. 4c indicates the architecture of the decomposition unit consisting of two modes corresponding to Fig. 2. A small Rom is available for *exp* and *ln* constant discretization factors in equation (7) and (8). To limit the resource on-chip for the Rom, an optimization strategy is utilized that $ln(2^a)$ coefficients can be halved as $ln(2^{-|a|})$ can be expressed as $-ln(2^{|a|})$. In addition, COMPACT offers a variety of complex arithmetic operations and integrated NAFs-based operations (see Table I). Those operations can be easily replaced by the simple arithmetic one, but it can effectively reduce the number of instruction entries and perform more operations in the same occupancy of the instruction ram. Furthermore, multiple operations executed together will achieve better performance compared to those executed in serialized.

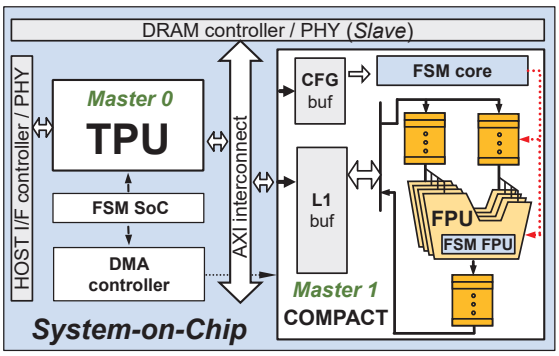
Type	Instruction	Levels of accuracy
Simple Arithmetic Ops.	$r = a + b$	No precis. loss
	$r = a - b$	No precis. loss
	$r = a \times b$	No precis. loss
	$r = a/b$	No precis. loss
	$r = e^x$	4
	$r = ln(x)$	2
Complex Arithmetic Ops.	$r = e^{-x}$	4
	$r = e^{2x}$	4
	$r = 1 + x$	No precis. loss
	$r = 1 - 2x$	No precis. loss
Non-linear Activation Funcs.	$r = \frac{1}{1+e^{-x}}$	4
	$r = 1 - \frac{2}{1+e^{2x}}$	4

TABLE I: COMPACT arithmetic instructions and the corresponding accuracy levels

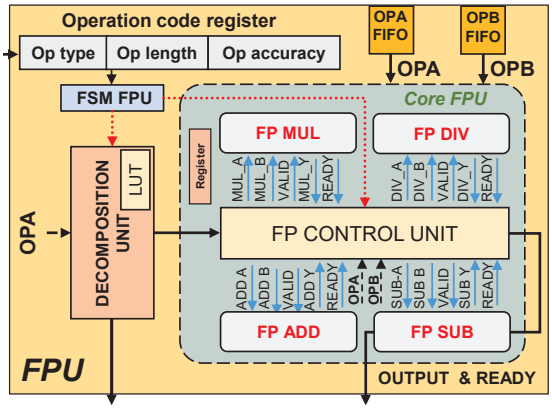
Unlike most conventional vector architectures, we adopt two levels of storage instead of a cache to explicitly transfer data with DDR. The details of memory are completely open and programmers can manually adjust the generated code for faster data access. For the "memory wall" is the major challenge in energy effectiveness, the frequent data movement between DDR and Compact may lead to significant overhead. A set of registers in FPUs are utilized for the storage of intermediate results produced by some arithmetic operations, such as *exp* and *ln*. Therefore, the overflow of storage during execution can be relieved, avoiding unnecessary data access with DDR.

V. Experimental Results

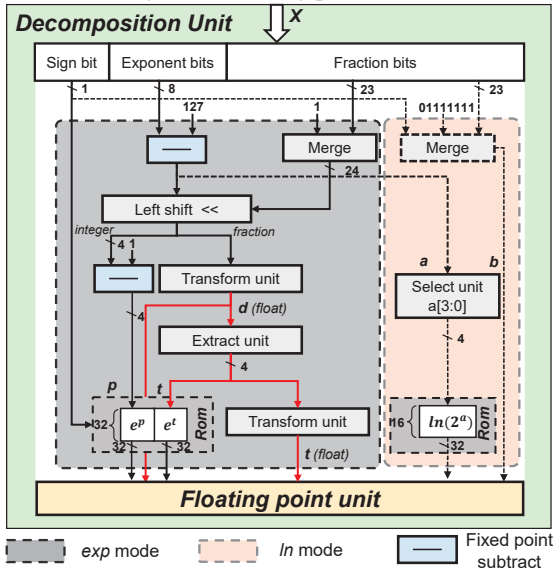
In this section, we evaluate the performance metrics of different NAFs in the FP32 data format. The execution of sigmoid is typically viewed as a basic metric in related works for its



(a) System-on-Chip including TPU and COMPACT



(b) Augmented floating point unit (FPU)



(c) Decomposition unit for *exp* and *ln* operations (refer to Fig. 2)

Fig. 4: The architecture of COMPACT in the System-on-Chip

universality. Consequently, we compare COMPACT with state-of-the-art methods from the high accuracy order to the lower order. The latency of other NAFs at different accuracy levels is also analyzed to exhibit precision adjustability. Finally, we list the overall distribution of errors at the highest precision level as a general evaluation. The architecture was implemented in Verilog, and the prototype was deployed on a Xilinx Kintex-7 410t FPGA for comprehensive evaluation. Note that, for a fair comparison, the experiment results are obtained at the lowest configuration—i.e. a single channel that is a single FPU case. In addition, the size of the input data is enough to offset the prologue and epilogue in vector operation, and the number of sampling points N is set as 1E4, equally spaced within $(-16, 16)$.

Table II lists the E_{max} of the sigmoid and the corresponding resource utilization and performance under different designs. For the design of Hajduk [18], we selected the McLaurin series (version B) and the Padé polynomial version (version J). For Pan [16], the NRA method was chosen. In the high-precise case, COMPACT achieves a more careful error control which is reduced by 26% compared with Hajduk (Padé), while the speed is improved by 30%. The improvement proves the effectiveness of the further argument reduction in some way, including the benefits of SIMD operations. For the middle accuracy level, i.e. the order of E$-$6 to E$-$4, our design still maintains a competitive advantage. Limited by the implementation of FP units, COMPACT has a lower frequency compared to the other design deployed in FXP format. However, it can be improved with the aid of a dedicated FP unit. When it comes to the low precision level, the PWL approach seems to be more efficient, but the implementation of NAFs on this level is only suitable for some scenarios insensitive to accuracy. From an application point of view, an essential difference between COMPACT and the other designs is the various precision levels are integrated. In other words, the adjustment from the order of E$-$8 to the order of E$-$3 is run-time, leading to a 46% reduction in the latency. Furthermore, our design has a wider AR attributed to the decomposition method mentioned in Section III.

Table III shows the latency of other NAFs at different E_{max} levels. As mish and tanhExp have multiple configuration options under the same order, we chose the most efficient one for comparison. Due to a correlation in the formula, tanh is similar to sigmoid in terms of accuracy and speed. For mish and tanhExp, there is a rapid increment in latency as the nested *exp* and *ln* operations. Corresponding to the long latency, relaxing the accuracy constraints can also lead to a significant gain in performance. For instance, the latency of mish and tanhExp will be reduced by 240ns and 120ns respectively, when the precision level is adjusted from the level of E$-$5 to E$-$4. In conclusion, NAFs including more precision adjustable operations can achieve better improvement from the relaxation of precision constraint.

Fig. 5 displays the error distributions of the four NAFs with the highest accuracy under our design. The error exploded mostly occurred with the input domain greater than the valid AR. During the execution of sigmoid and tanh, the input domain of *exp* won't be extended, therefore the errors are not magnified under the valid AR. In contrast, overflow is prone to occur for mish and tanhExp, due to the nested *exp* operations. However, errors are gradually stable as the input x is farther away from the origin. Tanh as the basis of mish and tanhExp tends to a fixed value with the increase of the input, which eliminates the error caused by overflow.

VI. Conclusion

This work presents COMPACT, a precision-adjustable co-processor for multiple NAFs. Taking advantage of a decomposition-based approximation engine, COMPACT works

	Hajduk (McLaurin) [18]	Hajduk (Padé) [18]	Ours (level 4)	Tiwari [19]	Pan (NRA) [16]	Ours (level 3)	Ours (level 2)	Wei [20]	Tsmots [14]	Ours (level 1)
Accuracy	high	high	high	middle	middle	middle	middle	low	low	low
E_{max}	1.192E-7	1.192E-7	8.84E-8	4.77E-5	5.72E-4	2.75E-6	8.86E-5	1.25E-2	1.85E-2	2.11E-3
Format	FP32	FP32	FP32	FXP32	FXP16	FP32	FP32	FXP16[in] FXP12[out]	FXP16	FP32
Latency (ns)	940.8	494.4	350	2130(+in) 1590(-in)	85	280	220	9.856	27	190
Frequency (MHz)	89.3	97.1	100	N/A	200	100	100	N/A	N/A	100
Resources (FPGA)	1916 LUT 792 FF 4 DSP	2624 LUT 1059 FF 8 DSP	2735 LUT 1163 FF 7 DSP	1388 LUT 597 FF 22 DSP	351 LUT 325 FF 6 DSP	2735 LUT 1163 FF 7 DSP	2735 LUT 1163 FF 7 DSP	140 LUT 23 FF 0 DSP	N/A	2735 LUT 1163 FF 7 DSP
Approxi. range (AR)	(-10, 10)	(-10, 10)	(-16, 16)	(-8, 8)	(-8, 8)	(-16, 16)	(-16, 16)	(-5, 5)	(-8, 8)	(-16, 16)

TABLE II: Benchmarking with state-of-the-art implementations on sigmoid NAF

E_{max}	sigmoid (ns)	tanh (ns)	mish (ns)	tanhExp (ns)
10^{-8}	350	N/A	N/A	N/A
10^{-7}	N/A	360	N/A	N/A
10^{-6}	280	290	1070	630
10^{-5}	220	N/A	930	560
10^{-4}	N/A	230	690	440
10^{-3}	190	200	660	410

TABLE III: Latencies of NAFs on different E_{max} levels

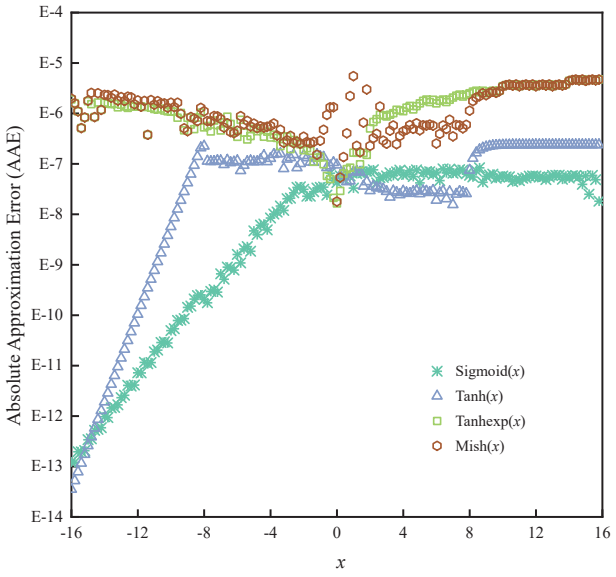


Fig. 5: Absolute approximation errors of implemented NAFs

together with its framework to efficiently process a variant of NAFs and explore trade-offs between precision level, fitting range, and execution latency. An FPGA prototype executing SIMD instructions demonstrates the design that outperforms relative works in multiple design metrics.

ACKNOWLEDGMENT

This work was funded by the Key-Area Research and Development Program of Guangdong Province (Grant No. 2019B010155003), National Natural Science and Foundation of China (NSFC 61902355), the Guangdong Basic and Applied Basic Research Foundation (Grant No. 2020B1515120044).

REFERENCES

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, and G. A. E. Al, "In-datacenter performance analysis of a tensor processing unit," in *Computer architecture news*, pp. 1–12, 2017.
[2] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," 2017.
[3] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.
[4] Yolo-v5. https://docs.ultralytics.com/, 2020.
[5] X. Liu and X. Di, "Tanhexp: A smooth activation function with high convergence speed for lightweight neural networks," *IET Computer Vision*, vol. 15, no. 2, pp. 136–150, 2021.
[6] A. Sherstinsky, "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, 2020.
[7] H. Zhang, W. Wu, Y. Ma, and Z. Wang, "Efficient hardware post processing of anchor-based object detection on fpga," in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 580–585, IEEE, 2020.
[8] Y. Peng, Y. Yang, Z. Wang, and C. Chen, "Anchorcapsule: a datastream-serving post-processor for object detection in embedded vision soc," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2022.
[9] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi gf100 gpu architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, 2011.
[10] Z. Hajduk, "High accuracy fpga activation function implementation for neural networks," *Neurocomputing*, vol. 247, pp. 59–61, 2017.
[11] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 802–815, IEEE, 2019.
[12] C. Brezinski and J. Van Iseghem, "Padé approximations," *Handbook of Numerical Analysis*, vol. 3, pp. 47–222, 1994.
[13] B. Pasca and M. Langhammer, "Activation function architectures for fpgas," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 43–437, IEEE, 2018.
[14] I. Tsmots, O. Skorokhoda, and V. Rabyk, "Hardware implementation of sigmoid activation functions using fpga," in *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*, pp. 34–38, IEEE, 2019.
[15] S. Gomar, M. Mirhassani, and M. Ahmadi, "Precise digital implementations of hyperbolic tanh and sigmoid function," in *2016 50th Asilomar Conference on Signals, Systems and Computers*, pp. 1586–1589, IEEE, 2016.
[16] Z. Pan, Z. Gu, X. Jiang, G. Zhu, and D. Ma, "A modular approximation methodology for efficient fixed-point hardware implementation of the sigmoid function," *IEEE Transactions on Industrial Electronics*, vol. 69, no. 10, pp. 10694–10703, 2022.
[17] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8612–8620, 2019.
[18] Z. Hajduk, "Hardware implementation of hyperbolic tangent and sigmoid activation functions," *Bulletin of the Polish Academy of Sciences. Technical Sciences*, vol. 66, no. 5, 2018.
[19] 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.
[20] L. Wei, J. Cai, V. Nguyen, J. Chu, and K. Wen, "P-sfa: Probability based sigmoid function approximation for low-complexity hardware implementation," *Microprocessors and Microsystems*, vol. 76, p. 103105, 2020.