Activations in Low Precision with High Accuracy

Tom Hubrecht, Orégane Desrentes, Florent de Dinechin

Activations in Low Precision with High Accuracy

Tom Hubrecht
École Normale Supérieure
Paris, France
ORCID: 0009-0002-3147-7736

Orégane Desrentes
Kalray S.A., CITI
Montbonnot, France
ORCID: 0009-0000-4012-9122

Florent de Dinechin
INSA Lyon, Inria, CITI, UR3720
Villeurbanne, France
ORCID: 0000-0003-4927-3301

***Abstract*—As machine learning hardware uses ever smaller number formats, this article surveys simple and effective techniques for the implementation of activation functions in low precision (fewer than 16 bits) with high accuracy. The implementation combines a fixed-point centric approach, efficient function-specific range reduction techniques, and state-of-the-art polynomial approximation. The resulting trade-offs are studied on both FPGA and ASIC. Functions considered in this article include tanh, sigmoid, ReLU variants such as GELU, ELU, SiLU, and exp for stable softmax, but the methodology can apply to more functions. These techniques are implemented in an open-source hardware generator that produces readable synthesizable VHDL.**

***Index Terms*—activation function, low precision, fixed point, tanh, sigmoid, ReLU, GELU, ELU, SiLU, softmax**

I. Introduction

The bibliographies of two recent surveys on activation functions (AF) for machine learning [1], [2] have more than 150 and 600 entries respectively. The present works focuses on *hardware* implementations of such functions, which has also been the subject of numerous articles, including [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]. The purpose of this paper is to improve upon all of these, by providing a combination of features that, to our knowledge, none offers: 1/ full flexibility, in terms of AF, of choice of implementation technique, and precision; 2/ uncompromising, consistent accuracy: as good as the output format allows; 3/ implementation techniques building upon a state of the art refined over decades. This methodology is implemented in an open-source tool called ALPHA (Activations in Low Precision with High Accuracy)[1]. ALPHA is a push-button AF hardware generator based on the open-source FloPoCo software [20], which was chosen because it is currently the best repository of generic fixed-point approximation techniques.

The trend in ML is to use ever smaller weights, currently down to 4 bits [21] or even 1.57 bits [22]. This leaves the precision of internal computations as the main goal to optimize. This work therefore focuses on fixed-point and low precision (fewer than 16 bits). A tool like ALPHA should allow ML researchers to focus on ML issues: Which AF function performs better? What is the smallest precision that can be exploited?

[1]Anonymous git: https://github.com/Activation-Functions/flopoco

A. Motivation: common issues in the state of the art

a) Lack of flexibility: Most previous AF hardware papers propose one technique for one function for a handful of precisions. Some [6] try to cover many functions but lack flexibility in other aspects.

b) Incomplete error assessment: Most papers build an architecture and report the corresponding error, e.g. maximum absolute error (MAE), average absolute error (AAE). With a handful of exceptions [4], [6], [17], [7], they only study *approximation* error, ignoring *rounding* errors due to the hardware. The latter typically reduces the accuracy by one or two bits, which cannot be neglected for an 8-bit output format. Some [8], [16] dispense with error assessment altogether and report ML metrics (e.g. top-1 accuracy on standard benchmarks). This is valid, but not informative in showing that the proposed AF implementation is a good choice at all.

c) Larger-than-needed datapaths: Implementations almost always output results which are vastly less accurate than their format allows. Let us just consider two recent examples. In [17] the output is a 16-bit number with 14 fractional bits, or a resolution of $2^{-14} \approx 0.00006$. Yet the proposed architecture is measured with a MAE of 0.025 ($> 2^{-6}$) and a AAE of 0.0049 ($> 2^{-8}$). This means that this hardware computes between 5 and 7 meaningless bits. In I-BERT [13] a 32-bit integer-only approximation to GELU is designed with a worst-case errors of 0.018, which is larger than 2^{-6}. Considering the integer bits of the format, this corresponds to 8 correct bits out of 32. This accuracy is obtained at the cost of 3 32-bit multiplications and a few other operations. ALPHA achieves the same accuracy using a very small table ($2^7 \times 8$ bits) and no arithmetic. Or, with only two much smaller multiplications (7x7 and 11x17 bits), it achieves a much higher accuracy (MAE smaller than 2^{-13}) on a 16-bit format. A key idea here is that accuracy should be correlated to the precision [20].

d) Ignorance of the state of the art outside the ML community: The massive ML bubble tends to shadow previous research in other communities: in the present case, signal processing and computer arithmetic researchers have worked for decades on hardware implementation of elementary functions. There exists handbooks on the subject [23], [24], [20] and many generic techniques have been developed [25], [26], [27], [28], [29], [30], [31], [32] that could be used for AFs: in the previous GELU example, ALPHA takes its small multipliers from the state of the art [30], [32].

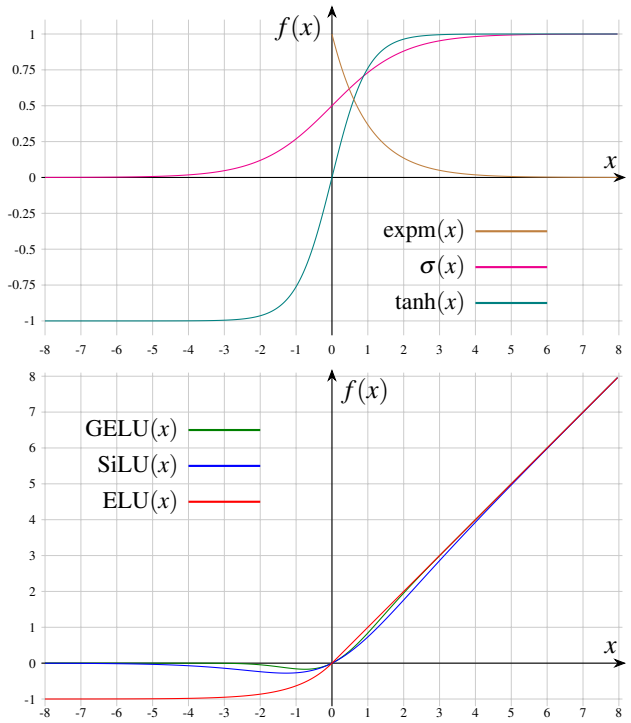


Fig. 1: Plots of the functions studied here

B. Outline

This article intends to address all these pitfalls. For this purpose, Section II lists useful mathematical properties of the functions that can be turned into efficient range reductions. Section III analyses in detail their fixed-point behaviour to avoid corner-case issues and define a strict accuracy specification. Section IV discusses the hardware implementation techniques used. Section V illustrates on ASIC and FPGA the trade-offs offered by the flexibility of ALPHA.

II. Analysis of function properties

a) Functions of interest: The functions currently implemented in ALPHA are plotted in Figure 1:

- Hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$
- Rectified Linear Unit: $\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$
- Exponential Linear Unit: $\mathrm{ELU}(x) = \begin{cases} -(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$
- Gaussian Error Linear Unit: $\mathrm{GELU}(x) = \frac{x}{2}(1 + \mathrm{erf}(\frac{x}{\sqrt{2}}))$
- Sigmoid Linear Unit (or Swish-1): $\mathrm{SiLU}(x) = x\sigma(x)$
- Exponential for stable softmax: $\mathrm{expm}(x) = e^{-x}$, $x \geq 0$

All these functions are evaluated by default on the input range $[-8, 8)$ (most commonly used in other papers). Most of the techniques advocated in this paper can easily be transposed to other intervals.

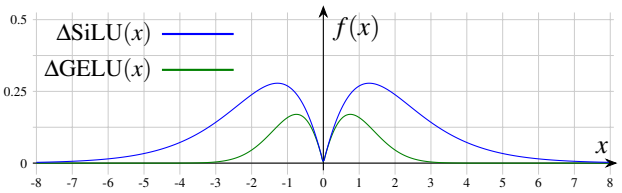


Fig. 2: Plot of the delta functions for the ReLU variants. For ELU, the delta function is expm (up to a symmetry).

b) Reduction to ReLU: SiLU and GELU were designed as continuous ReLU variant that help with training. Since SiLU and GELU are very similar to ReLU, the latter is a good first approximation to the function. Let us define

$$\Delta\mathrm{GELU} = \mathrm{ReLU} - \mathrm{GELU} \quad (1)$$

$$\Delta\mathrm{SiLU} = \mathrm{ReLU} - \mathrm{SiLU} \quad . \quad (2)$$

These functions, plotted in Fig. 2 capture all the non-linearity. The benefit is a smaller range, but also that the Δ functions are symmetric as shown in this figure, so they can be evaluated only on the positive domain. Altogether, for instance, for an 8-bit GELU implementation, a plain table needs $2^8 \times 8 = 2048$ bits, whereas ΔGELU only needs $2^7 \times 2 = 256$ bits.

c) Exploiting symmetries: There are also symmetries in tanh and σ (see Figure 1). These are commonly exploited to reduce both their range and their domain by one bit.

The parity of ΔGELU and ΔSiLU already discussed is not exploited to our knowledge so far, but we expect our readers to point us to papers that escaped our attention.

However, as shown in Figure 5, exploiting symmetries and reduction to ReLU costs extra adders. There is a trade-off here. Its net effect will be studied quantitatively in Section V.

III. Accuracy specification

This part assumes for clarity the same input and output width, w. ALPHA, however, supports different widths, and for instance the Δ functions have wider inputs than outputs.

A. Fixed-point numbers

In a fixed-point number, each bit has a fixed weight 2^i. The natural integer i is called the *position* of the bit. Bits with positive i belong to the integer part of a number, bits with strictly negative i belong the its fractional part (Figure 3).

A fixed-point format is characterized by 3 parameters: its signedness (whether a sign bit is present), the position m of its Most Significant Bit (MSB), and the position ℓ of its LSB. For signed numbers m is the position of the sign bit. The size (or width) of a fixed-point number is $w = m - \ell + 1$.

Two examples are depicted in Figure 3 for the input range $[-8, 8)$. This range determines $m = 3$, and for a total of $w = 8$ bits this implies that $\ell = m + 1 - 8 = -4$. For a 16-bit format we still have $m = 3$, but we have more fractional bits: $\ell = -12$. In short, m determines the range, and ℓ determines the precision of the format.

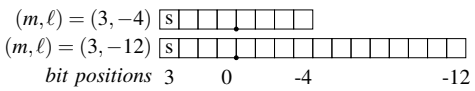


Fig. 3: 8- and 16-bit fixed-point formats for $X \in [-8,8)$

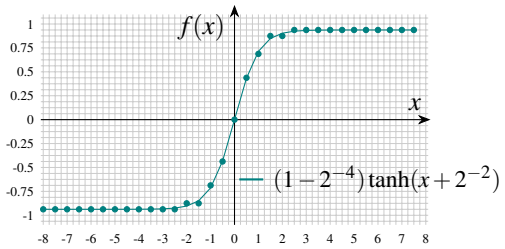


Fig. 4: Scaled tanh for $w = 5$ bits, with the CR values

The set of signed representable numbers is $\{-2^m, -2^m + 2^\ell, \ldots, 0, \ldots, 2^m - 2^\ell\}$. It is important to note that this set is not symmetrical. For instance the set of 8-bit fixed-point numbers covering our input range $[-8,8)$ is $\{-8, -7.9375, \ldots, -0.0625, 0, 0.0625, \ldots, 7.9375\}$.

B. Rescaling for fixed-point

As Figure 1 illustrates, all the functions have signed inputs except expm. We have a problem on the output of the three functions of Figure 1, top: they reach the value 1, which either is not representable, or requires one bit in the format that will only be used only for this value. The solution adopted in this work is a slight output scaling (which could in some cases be compensated by an inverse scaling in subsequent layers, but this is probably useless). Specifically, for $f \in \{\tanh, \sigma, \text{expm}\}$ we actually implement $f_s(x) = (1 - 2^{\ell_{\text{out}}}) f(x)$.

If we are to exploit the fact that tanh is odd ($\tanh(-x) = -\tanh(x)$), we have another problem with the lower bound -8 of our interval: its opposite is not representable. Attempting to compute it with the classical technique (binary negation then adding 2^ℓ) we will compute that -(-8) is 0, which may lead to a very wrong result if not managed properly. A naive solution is to add hardware that manage this case. This will cost a wide AND to detect -8, and a multiplexer to return the proper value in this case. For very small precisions it may negate the benefit of symmetry.

Another solution is simply to live with it. Indeed, the input to an AF is the output of a sum of products that should never overflow, either by training or by saturating the sum. For the same price it will be easy to make sure that the sum never reaches -8. In other words, if we worry for the value -8, we should even more worry about overflowing it, which is catastrophic as it turns a large negative into a large positive.

Luckily, there is no such problem for ΔSiLU and ΔGELU, the two other functions for which we want to exploit symmetry. They share the property that $\delta F(-8) \approx \delta F(0)$ (see Fig. 2). A naive computation of the opposite of -8 on $w-1$ bits will return 0, therefore the architecture of Figure 5c always work.

TABLE I: Fixed-point parameters for w output bits

f	input	output	symm.	m_{out}	ℓ_{out}	scaled
tanh	signed	signed	yes	0	$-w+1$	yes
σ	signed	unsigned	yes	-1	$-w$	yes
ReLU	signed	unsigned	no	3	$-w+4$	no
GELU	signed	signed	no	3	$-w+4$	no
ΔGELU	signed	unsigned	yes	-2	$-w+4$	no
SiLU	signed	signed	no	3	$-w+4$	no
ΔSiLU	signed	unsigned	yes	-3	$-w+4$	no
ELU	signed	signed	no	3	$-w+4$	no
ΔELU	unsigned	unsigned	no	0	$-w+4$	no
expm	unsigned	unsigned	no	-1	$-w$	yes

C. Specification for high accuracy functions

In this paper, we use a tight specification that is standard in the computer arithmetic community. Let f be the mathematical function under consideration and let F its hardware implementation. For a given input X, the absolute error is defined as $|F(X) - f(X)|$. Let $u = 2^{\ell_{\text{out}}}$ be the weight of the least significant bit of the output.

An implementation F is said to be *correctly rounded* (CR) if $\forall X \ |F(X) - f(X)| \leq u/2$. An equivalent formulation is that $F(X)$ is the machine number closest to the value of the function $f(X)$: This is the best one may get given that not all $f(X)$ are representable as machine numbers.

An implementation F is said to be *faithful* or *last-bit accurate* (LBA) if $\forall X \ |F(X) - f(X)| < u$. For reasons too long to describe here [20], LBA is much cheaper to achieve than CR as soon as the implementation involves an approximation. It still ensures that accuracy improves predictably with precision, and that all the output bits hold useful information.

All the ALPHA AFs based on plain tabulation ensure CR. Besides, computing the opposite of a fixed-point number is exact (except for 2^m) so table-based methods exploiting symmetries are also CR. All the methods involving an approximation ensure LBA. This shared specification makes for a fair comparison of implementations that provide the same application-level accuracy in Section V.

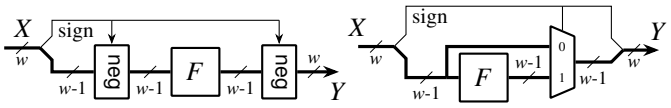
D. Exploiting bordercase quantization

Some quantization effects happen only in very small precisions. They can be exploited to achieve simpler architecture, but also show the practical limits of the ReLU variants. It is unclear if they can be exploited in the learning phase.

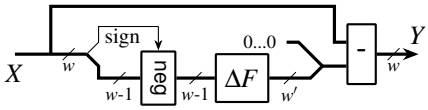
a) GELU is identical to ReLU for $w \leq 5$: In Figure 2, one may observe that $\forall x \ \Delta\text{GELU}(x) < 2^{-2}$. It therefore gets rounded to zero as soon as $\ell_{\text{out}} \geq -1$, with $\ell_{\text{out}} = -w+4$ (Table I).

b) SiLU is identical to ReLU for $w \leq 4$: idem.

c) GELU vanishes for $x > 4$ for $w \leq 15$: Figure 2 suggests that $\text{GELU}(x)$ and $\Delta\text{GELU}(x)$ vanish for $x > 4$. Indeed, $\Delta\text{GELU}(4) \approx 1.267e-4 < 2^{-12}$. Therefore, for $w \leq 15$, the function can be evaluated only on the interval $[-4,4)$. Note that for plain tables, synthesis tools should take care of this optimization: the current code does not handle it.



(a) Exploiting odd symmetry (b) Implementing ELU



(c) Reduction to ReLU (RRR) with even symmetry on ΔF

Fig. 5: Implementation techniques

IV. Hardware implementation techniques

a) Plain tabulation: This technique should not be despised: it provides correct rounding, and it actually benefits from the massive research investment in logic synthesis tools of the last decades, which allows them to scale to very large tables.

Note that the area unit of FPGAs is the LUTk (for Look Up Table with k inputs, and $k \in \{4,5,6\}$ for mainstream FPGA families). A tabulated function for $w = k$ costs exactly one LUTk per output bit. For $w = k+1$, it costs 2 or 3 LUTk per output bit (depending on the availability of a dedicated multiplexer in the FPGA cell). 8-bit functions can be implemented in less than 4 LUT6 per bit on modern FPGAs.

For instance in [4] the reported cost for 5-bit in, 6-bit out sigmoid is 25 LUT4. A plain table would consume 3 LUT4 per output bit, or a total of 18 LUT4 (and probably less if the synthesis tool finds any optimization to perform).

b) Differential compression: This is an exact compression method for tabulated functions which has the potential to reduce by up to 60% the number of bits stored at the cost of one addition [31], [10], [33]. In our study it becomes useful only for relatively large tables, for instance several of the Pareto points for a 10 bits sigmoid on FPGA uses symmetry and compression.

c) Piecewise polynomial approximation: Piecewise linear is the most used approximation technique [3], [5], [5], [8], [11], [12], [17] followed by second-order (spline) polynomials [34], [16]. ALPHA can do any degree but only uniform piecewise approximation [20] where all the subintervals have the same size: non-uniform techniques are not supported by FloPoCo (yet?). FloPoCo also sizes the multipliers to the minimum in a way that we have not seen in the AF literature.

One nice aspect of piecewise linear approximation is that the derivative is piecewise constant, which can be exploited to save memory during training [35].

V. Some results

This section attempts to expose the implementation space currently obtainable with the ALPHA tool. Raw tables and the corresponding plots are provided in appendix. Some of the table entries are N/A for not available: this is sometimes due to a bug of the tool, or to inacceptably long computation times, or simply that the entry doesn't make sense (e.g. reduction to RELU for 4-bit GELU or SiLU, see Sect. III-D).

TABLE II: Notations used in the results

T	plain table
PHd	approximation using a piecewise polynomial of degree d with Horner evaluation
C	tables are compressed [33]
Δ	reduction to ReLU is used
S	symmetry is exploited

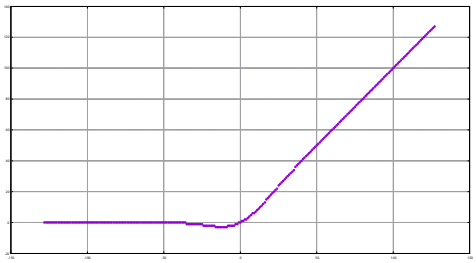


Fig. 6: Actual plot of an 8-bit implemented GELU

The main message is the following: there is no universally better technique, as the best solution depends on the function, on the precision, and on the desired cost/performance trade-off.

a) Synthesis context: All the results obtained in this paper were obtained: for FPGA, with Vivado 2024.1 for Kintex7 (xc7k70tfbv484-3), synthesis only (the operators are small enough that placing and routing them in an empty FPGA is not informative); for ASIC, with the Synopsys Design Compiler NXT for the TSMC N5 node 4nm technology. All the delays are reported in ns. The provided table and plots all use the notations of Table II.

b) Validation: Each function comes with a test bench, and an exhaustive test of the hardware, up to 16 bits, is a matter of seconds. The values being tested against are also provided in gnuplot form, an example of which is given in Figure 6.

c) Symmetry and compression versus synthesis tool optimization: Compression (C), reduction to ReLU (Δ) and symmetry (S) all involve the overhead of hardware additions. Tables IV to VIII show that this overhead is sometimes worth it, and sometimes not.

d) What is the best method?: All these results suggest that variants of tabulation are the best techniques to achieve high-quality implementations of activation functions for precision up to 10 bits. Piecewise linear approximation is the best choice for 16 bits. Second order approximation start to be relevant above 16 bits.

e) Which function is better in which context?: Figure 7 synthesizes relevant results from Tables VI to VIII to compare the area-delay trade-offs of the three ReLU variants. One may observe that ELU seems better both in area and delay, and this function should be preferred if it enables comparable results in machine learning tasks. Similarly, Figure 8 seems to suggest that tanh is in general cheaper to evaluate than sigmoid. These results are specific to the FPGA used here, but ALPHA should

make it easy to perform this comparison in other FPGA or ASIC contexts.

Table III provides a comparable study in an ASIC context.

VI. Conclusion and future work

ALPHA is a work in progress. We do hope people will still find novel AFs, novel ideas to evaluate AFs, and we look forward to integrating them if they outperform existing techniques. Adding a function to ALPHA should be relatively straightforward as most of the hard work is performed by the underlying layers of software (FloPoCo, Sollya, MPFR). For instance, we also have experimental support for the derivatives of the functions studied in this article but we chose to leave them out for the sake of space.

The next step, of course, is to integrate ALPHA-generated activation function hardware in complete ML accelerators.

References

[1] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," *Neurocomputing*, vol. 503, pp. 92–108, 2022.

[2] V. Kunc and J. Kléma, "Three decades of activations: A comprehensive survey of 400 activation functions for neural networks," *preprint arXiv:2402.09092*, 2024.

[3] H. Amin, K. Curtis, and B. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEE Proceedings - Circuits, Devices and Systems*, vol. 144, pp. 313–317(4), 1997.

[4] M. Tommiska, "Efficient digital implementation of the sigmoid function for reprogrammable logic," *IEE Proceedings-Computers and Digital Techniques*, vol. 150, no. 6, pp. 403–411, 2003.

[5] D. Larkin, A. Kinane, V. Muresan, and N. O'Connor, "An efficient hardware architecture for a neural network activation function generator," in *Third International Symposium on Neural Networks (ISNN)*. Springer, 2006, pp. 1319–1327.

[6] S. M. Ho and H. K.-H. So, "Nncore: A parameterized non-linear function generator for machine learning applications in fpgas," in *International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017, pp. 160–167.

[7] B. Pasca and M. Langhammer, "Activation function architectures for fpgas," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 43–437.

[8] L. Li, S. Zhang, and J. Wu, "An Efficient Hardware Architecture for Activation Function in Deep Learning Processor," in *3rd International Conference on Image, Vision and Computing (ICIVC)*, 2018, pp. 911–918.

[9] C.-H. Chang, H.-Y. Kao, and S.-H. Huang, "Hardware implementation for multiple activation functions," in *International Conference on Consumer Electronics-Taiwan (ICCE-TW)*. IEEE, 2019, pp. 1–2.

[10] Y. Xie, A. N. J. Raj, Z. Hu, S. Huang, Z. Fan, and M. Joler, "A twofold lookup table architecture for efficient approximation of activation functions," *Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 12, pp. 2540–2550, 2020.

[11] S. R. Chiluveru, S. Chunarkar, M. Tripathy, B. K. Kaushik *et al.*, "Efficient hardware implementation of dnn-based speech enhancement algorithm with precise sigmoid activation function," *Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 11, pp. 3461–3465, 2021.

[12] S. Bouguezzi, H. Faiedh, and C. Souani, "Hardware implementation of tanh exponential activation function using FPGA," in *18th International Multi-Conference on Systems, Signals & Devices (SSD)*. IEEE, 2021, pp. 1020–1025.

[13] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-BERT: Integer-only BERT quantization," in *International conference on machine learning*. PMLR, 2021, pp. 5506–5518.

[14] V. Shymkovych, S. Telenyk, and P. Kravets, "Hardware implementation of radial-basis neural networks with gaussian activation functions on fpga," *Neural Computing and Applications*, vol. 33, no. 15, pp. 9467–9479, 2021.

[15] R. Pogiri, S. Ari, and K. Mahapatra, "Design and fpga implementation of the lut based sigmoid function for dnn applications," in *International Symposium on Smart Electronic Systems (iSES)*. IEEE, 2022, pp. 410–413.

[16] Y.-H. Huang, P.-H. Kuo, and J.-D. Huang, "Hardware-Friendly Activation Function Designs and Its Efficient VLSI Implementations for Transformer-Based Applications," in *5th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 2023, pp. 1–5.

[17] K. Liu, W. Shi, C. Huang, and D. Zeng, "Cost effective tanh activation function circuits based on fast piecewise linear logic," *Microelectronics Journal*, vol. 138, p. 105821, 2023.

[18] Z. Zou, C. Zhang, S. Chen, H. Kou, and B. Liu, "Integer Arithmetic-Based and Activation-Aware GELU Optimization for Vision Transformer," in *Conference of Science and Technology for Integrated Circuits (CSTIC)*, 2024.

[19] J. Kim, S. Kim, K. Choi, and I.-C. Park, "Hardware-efficient softmax architecture with bit-wise exponentiation and reciprocal calculation," *Transactions on Circuits and Systems I: Regular Papers*, 2024.

[20] F. de Dinechin and M. Kumm, *Application-Specific Arithmetic*. Springer, 2024.

[21] C.-C. Lin, W. Lu, P.-T. Huang, and H.-M. Chen, "A 28nm 343.5 fps/W Vision Transformer Accelerator with Integer-Only Quantized Attention Block," in *6th International Conference on AI Circuits and Systems (AICAS)*. IEEE, 2024, pp. 80–84.

[22] S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei, "The Era of 1-bit LLMs: All Large Language models are in 1.58 Bits," *arXiv preprint arXiv:2402.17764*, 2024.

[23] J.-M. Muller, *Elementary Functions, Algorithms and Implementation*, 3rd ed. Birkhaüser Boston, 2016.

[24] A. Omondi, *Computer-Hardware Evaluation of Mathematical Functions*. Imperial College Press, 2016.

[25] W. F. Wong and E. Goto, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers," *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 278–294, 1994.

[26] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 842–847, 1999.

[27] D.-U. Lee, A. A. Gaffar, O. Mencer, and W. Luk, "Optimizing hardware function evaluation," *IEEE Transactions on Computers*, vol. 54, no. 12, pp. 1520–1531, 2005.

[28] D.-U. Lee, P. Cheung, W. Luk, and J. Villasenor, "Hierarchical segmentation schemes for function evaluation," *IEEE Transactions on VLSI Systems*, vol. 17, no. 1, 2009.

[29] A. G. Strollo, D. De Caro, and N. Petra, "Elementary functions hardware implementation using constrained piecewise-polynomial approximations," *IEEE Transactions on Computers*, vol. 60, no. 3, pp. 418–432, 2011.

[30] F. de Dinechin, M. Joldeş, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2010.

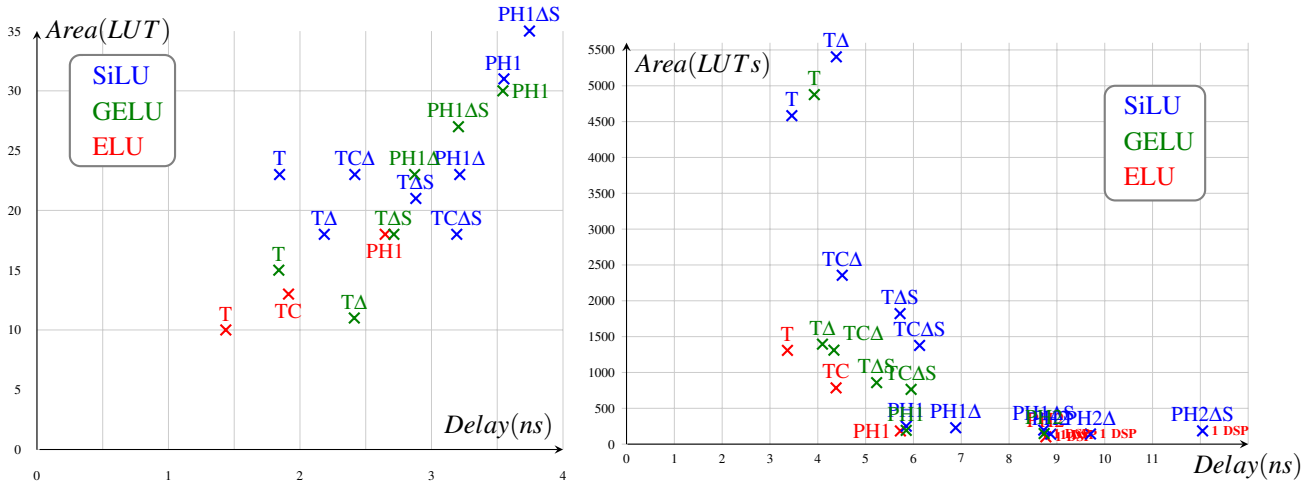
[31] S.-F. Hsiao, P.-H. Wu, C.-S. Wen, and P. K. Meher, "Table size reduction methods for faithfully rounded lookup-table-based multiplierless function evaluation," *Transactions on Circuits and Systems II*, vol. 62, no. 5, pp. 466–470, 2015.

[32] D. De Caro, E. Napoli, D. Esposito, G. Castellano, N. Petra, and A. G. Strollo, "Minimizing coefficients wordlength for piecewise-polynomial hardware function evaluation with exact or faithful rounding," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 5, pp. 1187–1200, 2017.

[33] M. Christ, L. Forget, and F. de Dinechin, "Lossless differential table compression for hardware function evaluation," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1642–1646, 2022.

[34] G. González-Díaz_Conti, J. Vázquez-Castillo, O. Longoria-Gandara, A. Castillo-Atoche, R. Carrasco-Alvarez, A. Espinoza-Ruiz, and E. Ruiz-Ibarra, "Hardware-based activation function-core for neural network implementations," *Electronics*, vol. 11, no. 1, p. 14, 2021.

[35] G. S. Novikov, D. Bershatsky, J. Gusak, A. Shonenkov, D. V. Dimitrov, and I. Oseledets, "Few-bit backward: Quantized gradients of activation functions for memory footprint reduction," in *International Conference on Machine Learning*. PMLR, 2023, pp. 26 363–26 381.



(a) precision 8 bits

(b) precision 16 bits

Fig. 7: Which RELU variant is cheaper on FPGA?

TABLE III: ASIC best results

w	tanh	σ	GELU	ELU	SiLU	expm
8	T 6.6 / 32	T 10.0 / 41	TΔ 3.0 / 42	TΔ 2.3 / 28	TΔS 4.1 / 49	T 9.0 / 35
12	PH1S 35.4 / 100	PH2 69.8 / 100	TΔS 17.8 / 89	PH1Δ 17.6 / 74	PH1ΔS 22.7 / 100	PH2 28.7 / 100
16	PH1S 146.8 / 100	PH2 99.0 / 100	PH1Δ S 55.7 / 100	PH1Δ 73.1 / 100	PH1ΔS 69.3 / 100	PH2 58.8 / 100

Each entry reads: best method, area in μ^2 / % of cycle

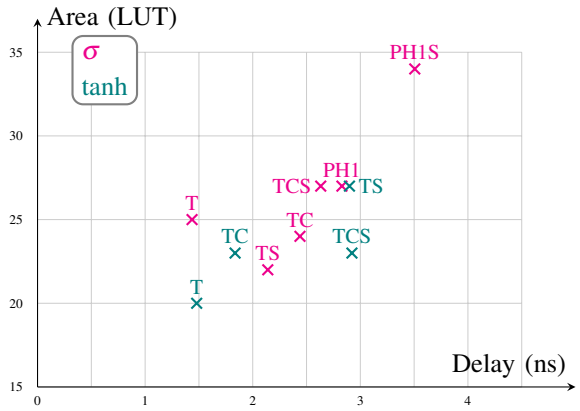
TABLE IV: FPGA synthesis results for σ

	4	6	8	10	12	16
T	**4** L, **1.06** ns	**6** L, **1.06** ns	25 L, **1.44** ns	97 L, **2.23** ns	375 L, **2.39** ns	5345 L, **3.38** ns
TC	**4** L, **1.06** ns	**6** L, **1.06** ns	24 L, 2.44 ns	80 L, 2.89 ns	281 L, 3.25 ns	4791 L, 4.58 ns
TS	**4** L, **1.06** ns	**6** L, **1.06** ns	**22** L, 2.14 ns	100 L, 2.98 ns	336 L, 3.57 ns	2869 L, 5.09 ns
TCS	**4** L, **1.06** ns	10 L, 1.49 ns	27 L, 2.63 ns	74 L, 3.55 ns	163 L, 4.2 ns	2244 L, 6.1 ns
PH1	7 L, 1.84 ns	N/A	27 L, 2.83 ns	65 L, 3.96 ns	129 L, 5.0 ns	425 L, 5.68 ns
PH1S	7 L, 1.81 ns	N/A	34 L, 3.51 ns	**57** L, 4.52 ns	**113** L, 6.48 ns	**296** L, 7.56 ns
PH2	21 L, 4.11 ns	47 L, 5.15 ns	79 L, 7.06 ns	125 L, 7.78 ns	81 L/1 DSP, 8.26 ns	184 L/1 DSP, 9.3 ns
PH2S	23 L, 3.34 ns	53 L, 5.38 ns	96 L, 7.21 ns	125 L, 8.48 ns	96 L/1 DSP, 9.71 ns	188 L/1 DSP, 11.26 ns
PH3	61 L, 6.43 ns	66 L, 6.25 ns	167 L, 11.09 ns	104 L/1 DSP, 10.7 ns	175 L/1 DSP, 12.04 ns	178 L/2 DSP, 13.84 ns
PH3S	N/A	97 L, 7.66 ns	148 L, 10.54 ns	144 L/1 DSP, 11.93 ns	206 L/1 DSP, 12.96 ns	181 L/2 DSP, 14.77 ns

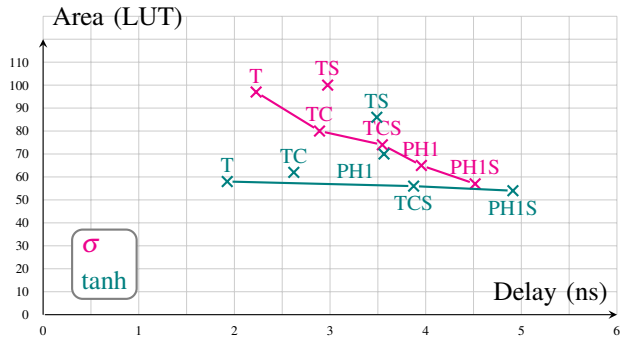
TABLE V: FPGA synthesis for tanh

	4	6	8	10	12	16
T	**3** L, **1.06** ns	**5** L, **1.06** ns	**20** L, **1.48** ns	58 L, **1.93** ns	233 L, **2.56** ns	4670 L, **3.5** ns
TC	**3** L, **1.06** ns	7 L, 1.5 ns	23 L, 1.84 ns	62 L, 2.62 ns	188 L, 2.93 ns	2662 L, 3.91 ns
TS	4 L, **1.06** ns	10 L, 1.83 ns	27 L, 2.9 ns	86 L, 3.49 ns	177 L, 4.13 ns	1917 L, 6.12 ns
TCS	4 L, **1.06** ns	10 L, 1.83 ns	23 L, 2.92 ns	56 L, 3.87 ns	199 L, 4.67 ns	1500 L, 6.73 ns
PH1	N/A	11 L, 1.92 ns	N/A	70 L, 3.56 ns	127 L, 3.81 ns	518 L, 5.65 ns
PH1S	N/A	16 L, 2.41 ns	N/A	**54** L, 4.91 ns	**96** L, 5.56 ns	**347** L, 8.41 ns
PH2	N/A	34 L, 3.65 ns	72 L, 5.45 ns	107 L, 6.99 ns	173 L, 7.49 ns	220 L/1 DSP, 9.57 ns
PH2S	N/A	47 L, 4.39 ns	68 L, 6.61 ns	118 L, 8.08 ns	159 L, 9.26 ns	234 L/1 DSP, 12.84 ns
PH3	41 L, 4.69 ns	45 L, 4.34 ns	136 L, 8.51 ns	200 L, 10.35 ns	128 L/2 DSP, 13.36 ns	192 L/2 DSP, 12.89 ns
PH3S	53 L, 5.36 ns	87 L, 6.29 ns	208 L, 11.01 ns	195 L, 11.37 ns	120 L/2 DSP, 14.67 ns	242 L/2 DSP, 15.97 ns

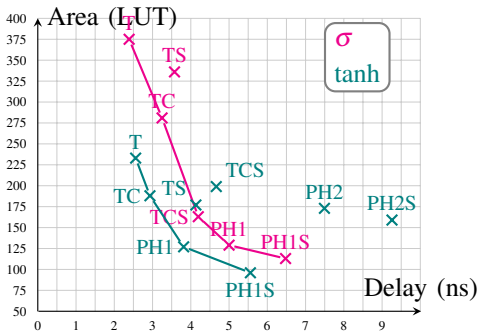
Remark: for 10 bits, the T or TCS variants should be preferred as they offer correct rounding where the PH1S variant is only faithful.



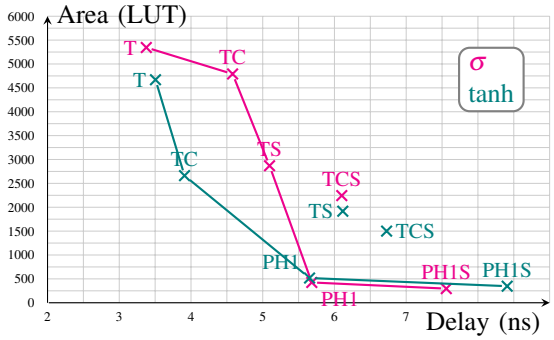
(a) 8 bits



(b) 10 bits

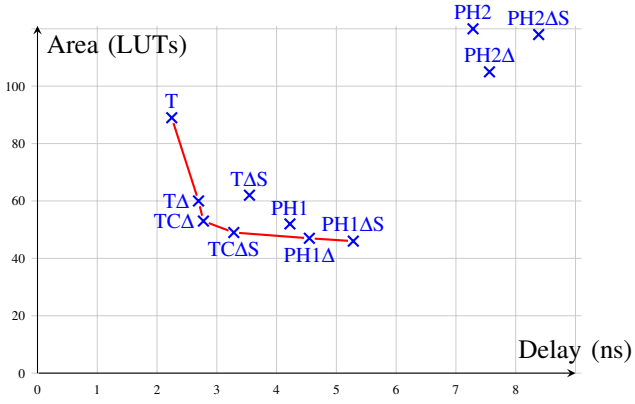


(c) 12 bits DSP-less solutions

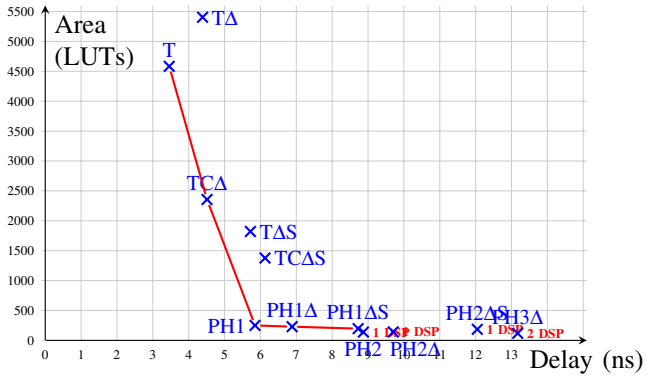


(d) 16 bits DSP-less solutions

Fig. 8: FPGA synthesis results for σ and tanh with pareto fronts



(a) 10 bits



(b) 16 bits

Fig. 9: FPGA synthesis results for SiLU with pareto front

TABLE VI: FPGA synthesis for GELU

	4	6	8	10	12	16
T	**3** L, **1.06** ns	**6** L, **1.06** ns	15 L, **1.84** ns	49 L, **2.16** ns	179 L, **2.51** ns	4877 L, **3.93** ns
TC	N/A	**6** L, **1.06** ns	15 L, **1.84** ns	49 L, **2.16** ns	179 L, **2.51** ns	4877 L, **3.93** ns
TΔ	N/A	N/A	**11** L, 2.41 ns	**33** L, 2.55 ns	117 L, 2.95 ns	1395 L, 4.1 ns
TΔS	N/A	N/A	18 L, 2.71 ns	**33** L, 3.34 ns	121 L, 3.5 ns	858 L, 5.23 ns
TCΔ	N/A	N/A	**11** L, 2.41 ns	34 L, 2.77 ns	97 L, 3.27 ns	1311 L, 4.34 ns
TCΔS	N/A	N/A	18 L, 2.71 ns	**33** L, 3.4 ns	85 L, 4.03 ns	764 L, 5.95 ns
PH1	**3** L, **1.06** ns	13 L, 1.92 ns	30 L, 3.54 ns	52 L, 4.22 ns	86 L, 4.78 ns	192 L, 5.85 ns
PH1Δ	N/A	N/A	23 L, 2.87 ns	45 L, 4.55 ns	75 L, 5.45 ns	N/A
PH1ΔS	N/A	N/A	27 L, 3.21 ns	44 L, 5.28 ns	**71** L, 6.55 ns	183 L, 8.19 ns
PH2	N/A	19 L, 2.99 ns	67 L, 5.85 ns	65 L/1 DSP, 8.33 ns	89 L/1 DSP, 8.51 ns	148 L/1 DSP, 8.73 ns
PH2Δ	N/A	N/A	50 L, 5.26 ns	66 L/1 DSP, 9.19 ns	90 L/1 DSP, 9.38 ns	151 L/1 DSP, 9.56 ns
PH2ΔS	N/A	N/A	53 L, 5.57 ns	83 L/1 DSP, 9.49 ns	94 L/1 DSP, 10.36 ns	**146** L/1 DSP, 10.96 ns
PH3Δ	N/A	N/A	129 L, 10.24 ns	87 L/2 DSP, 12.95 ns	86 L/2 DSP, 12.91 ns	164 L/2 DSP, 14.04 ns

Remark: For this function, compression of a plain table is never effective. In this case the tool produces an uncompressed table.

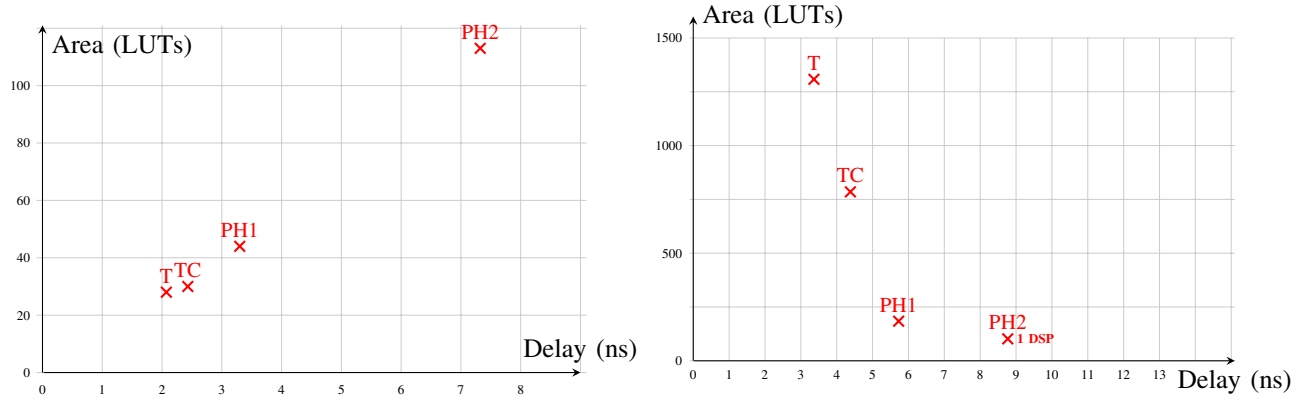
TABLE VII: FPGA synthesis for SiLU

	4	6	8	10	12	16
T	**3** L, **1.06** ns	**6** L, **1.06** ns	23 L, **1.84** ns	89 L, **2.25** ns	334 L, **2.6** ns	4583 L, **3.46** ns
TC	N/A	**6** L, **1.06** ns	23 L, **1.84** ns	89 L, **2.25** ns	334 L, **2.6** ns	4583 L, **3.46** ns
TΔ	N/A	**6** L, **1.06** ns	**18** L, 2.19 ns	60 L, 2.69 ns	206 L, 3.17 ns	5404 L, 4.39 ns
TΔS	N/A	**6** L, **1.06** ns	21 L, 2.88 ns	62 L, 3.55 ns	175 L, 3.47 ns	1821 L, 5.72 ns
TCΔ	N/A	**6** L, **1.06** ns	23 L, 2.42 ns	53 L, 2.78 ns	184 L, 3.53 ns	2356 L, 4.51 ns
TCΔS	N/A	**6** L, **1.06** ns	**18** L, 3.19 ns	49 L, 3.28 ns	119 L, 4.07 ns	1377 L, 6.13 ns
PH1	**3** L, **1.06** ns	18 L, 2.41 ns	31 L, 3.55 ns	52 L, 4.22 ns	87 L, 4.78 ns	250 L, 5.85 ns
PH1Δ	N/A	13 L, 2.44 ns	23 L, 3.21 ns	47 L, 4.55 ns	81 L, 5.42 ns	229 L, 6.88 ns
PH1ΔS	N/A	16 L, 2.42 ns	35 L, 3.75 ns	**46** L, 5.28 ns	73 L, 6.56 ns	196 L, 8.72 ns
PH2	11 L, 2.23 ns	49 L, 5.89 ns	94 L, 7.65 ns	120 L, 7.29 ns	**72** L/1 DSP, 8.04 ns	139 L/1 DSP, 8.87 ns
PH2Δ	N/A	33 L, 4.76 ns	77 L, 7.28 ns	105 L, 7.56 ns	144 L, 8.2 ns	142 L/1 DSP, 9.7 ns
PH2ΔS	N/A	35 L, 5.13 ns	85 L, 8.07 ns	118 L, 8.38 ns	145 L, 9.11 ns	185 L/1 DSP, 12.05 ns
PH3Δ	N/A	56 L, 6.25 ns	142 L, 9.48 ns	196 L, 11.39 ns	92 L/2 DSP, 13.05 ns	**118** L/2 DSP, 13.17 ns

Remark: For this function, compression of a plain table is never effective. In this case the tool produces an uncompressed table.

TABLE VIII: FPGA synthesis for ELU

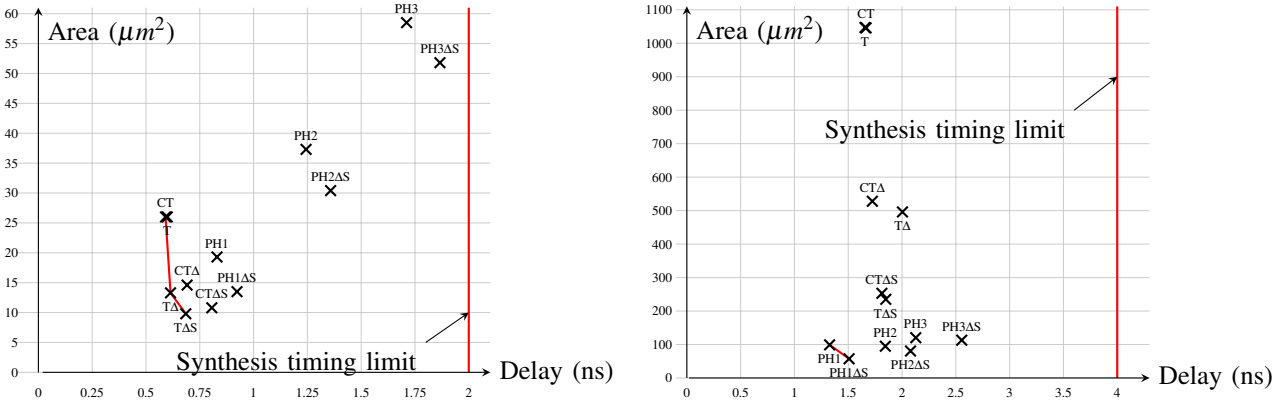
	4	6	8	10	12	16
T	**3** L, **1.06** ns	**5** L, **1.06** ns	**10** L, **1.44** ns	**28** L, **2.07** ns	97 L, **2.48** ns	1308 L, **3.37** ns
TC	**3** L, **1.06** ns	**5** L, **1.06** ns	13 L, 1.91 ns	30 L, 2.43 ns	68 L, 3.5 ns	785 L, 4.38 ns
PH1	**3** L, **1.06** ns	**5** L, **1.06** ns	18 L, 2.65 ns	44 L, 3.3 ns	**57** L, 5.04 ns	184 L, 5.73 ns
PH2	N/A	13 L, 2.11 ns	52 L, 5.12 ns	113 L, 7.32 ns	80 L/1 DSP, 8.71 ns	**102** L/1 DSP, 8.77 ns
PH3	N/A	53 L, 6.27 ns	134 L, 9.83 ns	213 L, 10.73 ns	80 L/2 DSP, 12.6 ns	122 L/2 DSP, 13.04 ns



(a) 10 bits

(b) 16 bits

Fig. 10: FPGA synthesis results for ELU



(a) 10 bits synthesised for 500 MHz

(b) 16 bits synthesised for 250 MHz

Fig. 11: ASIC Synthesis for SiLU