SimBU: Self-Similarity-Based Hybrid Binary-Unary Computing for Nonlinear Functions

Alireza Khataei, Gaurav Singh, Kia Bazargan
Department of Electrical and Computer Engineering
University of Minnesota
Minneapolis, MN, USA
{khata014, singh431, kia}@umn.edu

***Abstract*—Unary computing is a relatively new method for implementing arbitrary nonlinear functions that uses unpacked, thermometer number encoding, enabling much lower hardware costs. In its original form, unary computing provides no trade-off between accuracy and hardware cost. In this work, we propose a novel self-similarity-based method to optimize the previous hybrid binary-unary work and provide it with the trade-off between accuracy and hardware cost by introducing controlled levels of approximation. Looking for self-similarity between different parts of a function allows us to implement a very small subset of core unique subfunctions and derive the rest of the subfunctions from this core using simple linear transformations. We compare our method to previous works such as FloPoCo-LUT (lookup table), HBU (hybrid binary-unary) and FloPoCo-PPA (piecewise polynomial approximation) on several 8–12-bit nonlinear functions including Log, Exp, Sigmoid, GELU, Sin, and Sqr, which are frequently used in neural networks and image processing applications. The area × delay hardware cost of our method is on average 32%–60% better than previous methods in both exact and approximate implementations. We also extend our method to multivariate nonlinear functions and show on average 78%–92% improvement over previous work.**

***Index Terms*—hardware acceleration, approximate computing, unary computing, stochastic computing, table-based method, piecewise polynomial approximation, nonlinear function, activation function**

I. Introduction

BINARY representations have been the dominant data encoding scheme in digital systems for many years. Despite low memory requirements, performing computations in binary is not trivial due to the positional nature of the representation, which requires unpacking bits, performing computations such as partial product generation in a multiplication operation, and packing partial results through carry chain propagation into the final output. Additionally, the binary representation is not error-resilient, and flipping a single bit may introduce a significant amount of error based on the position of the bit [1].

Table-based methods are mostly used to implement nonlinear functions in hardware. Although they reduce latency compared to iterative approaches such as CORDIC [2], they suffer from the size of lookup tables, which grow exponentially as resolution increases. Therefore, they are usually paired with approximate methods [3]–[9] to reduce the size of lookup tables at the expense of accuracy. FloPoCo[1] [10] is a generator of arithmetic cores for FPGAs using a comprehensive set of methods. It has a tool to implement arbitrary functions without accuracy loss using pure lookup tables. Additionally, it has another tool to approximately implement functions using a piecewise polynomial approximation [3], in which the input interval of a function is divided into several subintervals addressed by the higher bits of the input, and each subfunction is approximated by a polynomial with the Horner scheme given a target maximum error. We refer to these two methods as FloPoCo-LUT and FloPoCo-PPA, respectively.

Pure unary (PU) [11], [12] was introduced as a way of implementing nonlinear functions by encoding the binary numbers in the form of unary codes and using a network of wires and XOR gates to perform the computations in the unary domain. In low-resolution computations—up to 12 bits—PU outperforms the conventional binary and stochastic computing methods [13]–[16] in terms of the area × delay hardware cost [12]. Despite the simplicity of scaling networks, converting data between binary and unary is costly, especially as the resolution increases, leading to exponentially higher hardware cost.

To reduce the hardware cost of PU, especially in non-monotonic functions at high resolutions, hybrid binary-unary (HBU) [17], [18] was proposed to take advantage of unary and binary to make the method more scalable. It breaks a function into subfunctions with limited input and output ranges and implements them efficiently based on the PU method. Therefore, the lower bits of the input compute the subfunctions in unary, and the higher bits complete the computations in binary. By limiting the input and output range of the function, this method dramatically reduces the binary-unary and unary-binary conversion costs, which in turn leads to lower area × delay hardware cost compared to PU.

Our previous work, approximate hybrid binary-unary (AHBU) [19], was recently proposed to reduce the hardware cost of HBU by sacrificing accuracy. It manipulates the higher bits of subfunctions and use very rudimentary self-similarity measures to reduce hardware cost. However, AHBU focuses on mean absolute error (MAE) as a metric for how much approximation tolerance it should have, which is a major flaw, because it does not consider the maximum error, which is a critical issue in approximate computing. Nonetheless, it can be deployed in machine learning applications where there is relatively high tolerance for errors.

[1]Available at http://www.flopoco.org.

In this paper, which is the extended version of [20], we propose a novel method to optimize the hardware cost of HBU for univariate and multivariate functions using self-similarities among subfunctions. Given a target maximum absolute error, it makes pairwise comparisons to figure out which subfunctions can be derived from each other through simple bitwise transformations. For instance, if a subfunction g_i is similar to the inverse of a subfunction g_j, then we can ignore the implementation of g_i in unary and use NOT gates to derive it from g_j. After finding the derivable subfunctions, our method tries to find the minimum set of "core" subfunctions that can derive all subfunctions. This practice replaces the unary subfunctions with simple post-processing logic gates to reduce the hardware cost.

For evaluation purposes, we implemented several univariate nonlinear functions at 8-, 10-, and 12-bit resolutions using our method and previous works including FloPoCo-LUT, HBU, and FloPoCo-PPA. Given the approximation of the least significant bit, our method improves the hardware cost of FloPoCo-LUT, HBU, and FloPoCo-PPA on average by 60%, 57%, and 35%, respectively. Additionally, if no approximation is allowed, which is a tough restriction on our proposed algorithm, our method still outperforms FloPoCo-LUT and HBU on average by 36% and 32%, respectively. Additionally, we implemented several *multivariate* nonlinear functions at 8-bit resolution using our method and previous HBU work. Our method improves the hardware cost of HBU on average by 92% given the approximation of the least significant bit and by 78% given no approximation. By implementing 10-bit homomorphic filtering and 8-bit Robert's cross edge detection, we also show that our method can implement both applications with no quality loss at lower hardware cost than previous works.

Although our method outperforms FloPoCo-LUT and HBU at higher resolutions, neither of these methods nor our method area efficient at resolutions beyond 12-bit [18], [21]. In such cases, other approximate methods might be more beneficial in terms of hardware costs. However, they provide trade-offs between accuracy, area, delay, and other aspects of hardware costs. Example of such methods include stochastic computing (SC) and bitstream processing [13]–[16], [22]–[30], piecewise polynomial approximation (PPA) [3], bipartite table (BT) [31], and multipartite table (MT) [32], [33] methods. For instance, SC methods are efficient in terms of area at higher resolutions, but they suffer from long latency and approximation errors. In addition, PPA, BT, and MT methods might deliver higher throughput than SC methods, but they all rely on lookup tables to evaluate a function. Although some techniques have been proposed in [34], [35] to compress the lookup tables in such table-based methods, the sizes of the lookup tables can potentially affect the area efficiency of these methods. As a result, each of these methods has their specific applications, and one should consider design requirements to choose the optimum solution.

Our contributions in this work include the following methods and results:

- Our method changes the way HBU breaks up functions. HBU breaks up functions into non-uniform input-range subfunctions, with the sole goal of reducing hardware cost. Our method forces an equal input range for all subfunctions, with the hope of deriving many of them from a core set of subfunctions. Our method uses a number of linear transformations to check if a subfunction can be derived from another one. This is in contrast to the self-similarity measures in AHBU [19] that only looked at non-transformed matches within the approximation tolerance.
- Instead of pairwise, local comparison between subfunctions as done in AHBU, our method uses a better optimization method to find the best subset of subfunctions to implement a function. For instance, using the lowest possible approximation error for implementing $f(x) = [1 + sin(2\pi x)] \div 2$, our previous AHBU work could reduce the total number of subfunctions from 512 to 218, whereas our new method can reduce them from 512 to 71 unique subfunctions.
- Our method provides HBU with a trade-off between accuracy in terms of maximum error and hardware cost in terms of area × delay.
- It outperforms the hardware cost of FloPoCo-LUT and HBU with no approximation error.
- It outperforms the hardware cost of FloPoCo-PPA at up to 12-bit resolutions using the same approximation error budget.

The rest of the paper is as follows. Section II-A reviews the basics of PU and HBU as fundamental constituents of our method. Section II-B introduces our proposed method and algorithm, followed by a guiding example in Section II-C. Section III extends our proposed method to multivariate functions. In Section IV, our method is compared to previous works on a number of nonlinear functions. The benefits of our method are evaluated in the implementations of homomorphic filtering and Robert's cross edge detection as image processing applications in Section V. Finally, Section VI concludes the paper and results.

II. Methodology for Univariate Functions

A. Previous Unary Works

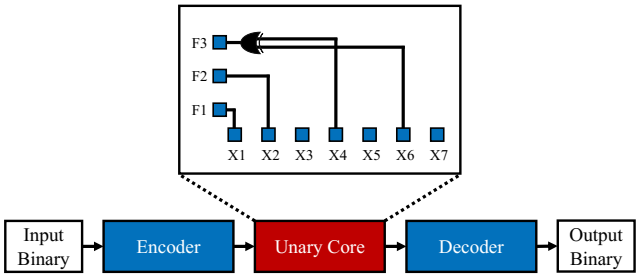
PU (pure unary) [12] is a method that implements a math function $f(x)$ using a network of wires and XOR gates, called "unary core". It converts the input binary to the unary domain in the form of thermometer codes, called "unary" codes. For a w-bit binary number, it uses $2^w - 1$ bits, in which the first m bits are 1's and the rest are 0's to represent the decimal value m out of the maximum value $2^w - 1$. For example, 011 in binary equals 1110000 in unary, and 100 in binary equals 1111000 in unary. After encoding, the input unary is mapped to output unary through a unary core, which is designed based on the output values of the function. Finally, the output unary is converted back to binary as the final output.

Fig. 1 shows the architecture of the PU method for an arbitrary function, described as lookup tables in binary and unary. In unary methods, the input and output values of a function are considered unsigned integers and then encoded to the unary domain, although one can change the interpretation

of the raw input/output values to represent signed integers or signed/unsigned fixed-point values. In this figure, the function has a 3-bit input binary and 2-bit output binary. Therefore, the input ranges from $x = 000$ to $x = 111$, and the output can range from $f = 000$ to $f = 111$. To represent the values in unary, we need to use 7 bits for the input and 3 bits for the output, since the maximum input and output are 7 and 3 in decimal, respectively. We represent the input unary as $X = X_1X_2X_3X_4X_5X_6X_7$ and output unary as $F = F_1F_2F_3$. In order to design the unary core, we need to explore the function's lookup table in unary step-by-step from $X = 0000000$ to $X = 1111111$. In this function, when $X = 0000000$, the output $F = 000$, but when the input $X = 1000000$, the output $F = 100$. By looking at the unary values, we can realize that X_1 triggers F_1. Therefore, X_1 is connected to F_1 through a wire in the unary core. Similarly, when the input $X = 1100000$, the output $F = 110$. As a result, X_2 is a triggering point for F_2, and they are connected through a wire. In contrast, if the input is increased to $X = 1110000$, the output is not changed, and it means X_3 is not a triggering point and it is not connected to any of the output pins. If the input is increased to $X = 1111000$, the output $F = 111$, and it means X_4 is a triggering point for F_3, and these two pins must be connected through a wire (ignore the XOR gate in Fig. 1 for now). If the input is increased to $X = 1111100$, the output remains unchanged, and therefore X_5 is an unconnected pin. Thus far, the function has been monotonically increasing. However, if the input is increased to $X = 1111110$, the output decreases to $F = 110$. In other words, X_6 has alternated the state of F_3, that has been triggered by X_4. For this reason, we need to connect X_4 and X_6 to F_3 through an XOR gate. The XOR gate handles the non-monotonic section of the function. As a result, $F_3 = 0$ when both X_4 and X_6 are the same, but $F_3 = 1$ when $X_4 = 1$ and $X_6 = 0$. It should be noted that $X_4 = 0$ and $X_6 = 1$ cannot happen at the same due to the unary representation which is based on a left-flushed thermometer encoding. Finally, X_7 is left unconnected, because X_7 is neither a triggering nor an alternating point, and it does not change the state of any output pins. The following equations shows the correspondence between the input and output unary bits.

$$F_1 = X_1,\ F_2 = X_2,\ \text{and}\ F_3 = X_4 \oplus X_6$$

The PU method is not scalable: as the width w of the input binary number increases, encoder and decoder units become exponentially larger, as they need to cover the range $\{0, 1, \cdots, 2^w - 1\}$. The HBU (hybrid binary-unary) method [18] addresses this issue by preserving the higher bits of input and output in binary, hence keeping the encoding scalable for these bits, and only converting the lower bits of the input into unary to perform efficient computations. The two parts are assembled into the final output encoded in binary. More specifically, the method breaks a function $f(x)$ into several subfunctions $g_i(x)$. Functions $g_i(x)$ are chosen so that they cover an output range from 0 to a number Max_i. This range would be less than or equal to the range of the original function $f(x)$, which means potentially smaller unary-to-binary encoders would be needed to convert the output



Binary

x	f
000	00
001	01
010	10
011	10
100	11
101	11
110	10
111	10

Unary

X	F
0000000	000
1000000	100
1100000	110
1110000	110
1111000	111
1111100	111
1111110	110
1111111	110

Fig. 1: Hardware architecture of PU [12] for an arbitrary function. The function's output values for all possible input values are shown in binary and unary.

back to binary. Each subfunction would be added to a bias value b_i to reconstruct the original function $f(x)$. The addition operation is performed in binary.

$$f(x) = \begin{cases} f_1(x) = b_1 + g_1(x) & x \in [0, x_1) \\ \cdots \\ f_n(x) = b_n + g_n(x) & x \in [x_{n-1}, x_n) \end{cases}$$

As mentioned above, in contrast to $f(x)$, a subfunction $g_i(x)$ has a limited input and output range; therefore, it can be efficiently implemented using the PU method [12], leading to a more scalable method. All these subfunctions are computed concurrently using the lower bits of the input x, but only one of them holds the relevant result. The higher bits of the input binary x determine which subfunction is going to be used and multiplexes the corresponding bias from a binary lookup table. Finally, a binary adder adds the subfunction and the bias together to compute the final output. Fig. 2 shows the overall architecture of HBU.

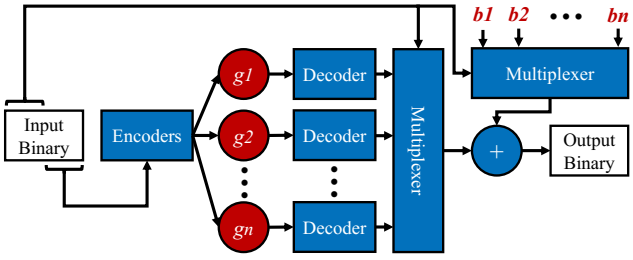


Fig. 2: Hardware architecture of HBU [18].

B. Our Proposed Work

We first present the formal explanation of how our method works through equations and pseudo-code in this section,

followed by a guiding example in Sec. II-C. Readers who are more comfortable with visual learning might want to first read that section and then come back to this section.

The function breaking in HBU is a complex process that uses many parameters, including subfunctions' slopes and output ranges to break a function hierarchically. The subfunctions might have different input ranges, leading to a set of binary-to-unary encoders with different input/output sizes. In contrast to HBU, we uniformly break a function $f(x)$ into $n = 2^{w_b}$ subfunctions and separate their initial biases (lines 5-9 of Algorithm 1). For a w-bit function, the input range of all the subfunctions is $w_u = w - w_b$ bits.

$$f(x) = \begin{cases} f_1(x) = b_1 + g_1(x) & x \in [0, 2^{w_u}) \\ ... & \\ f_n(x) = b_n + g_n(x) & x \in [(n-1) \times 2^{w_u}, n \times 2^{w_u}) \end{cases}$$

By uniformly breaking the function, only one encoder is needed to convert the w_u lower bits from binary to unary, whereas the same does not hold in HBU. The novelty in our method is to measure pairwise similarities between subfunctions (lines 11-19 of Algorithm 1) and find the minimum set of unique subfunctions, from which all the other subfunctions can be derived using a set of bitwise transformations (lines 21-39 of Algorithm 1). For instance, if $g_i(x)$ is similar to the inverse of $g_j(x)$, then we can implement $g_j(x)$ and use NOT gates to derive $g_i(x)$ from it.

Definition: The approximation error of deriving $g_i(x)$ from $g_j(x)$ through a transformation T_i is defined as:

$$Err(g_i, T_i, g_j) = max\{|Fixed(b_i + bm_i + T_i\{g_j(x)\}) - Float(b_i + g_i(x))|\} \quad (1)$$

where $Fixed(x)$ and $Float(x)$ denote the fixed-point and floating-point values of the unsigned integer x, respectively. T_i is a bitwise transformation, that can include an inversion, right shift, and left shift, and bm_i is a constant that modifies the vertical position of $T_i\{g_j(x)\}$ to reduce the approximation error, and it can be obtained by Eq. 2

$$bm_i = \lfloor \frac{1}{2^{w_u}} \sum_x (g_i(x) - T_i\{g_j(x)\}) \rceil \quad (2)$$

where $\lfloor \rceil$ denotes rounding to the nearest integer. Intuitively, bm_i tries to "center" the transformed subfunction around the center of gravity of the target subfunction to reduce the amount of error in deriving the subfunction. This constant cannot take any arbitrary value and is constrained by Eq. 3:

$$0 \leq b_i + bm_i + T_i\{g_j(x)\} < 2^w \quad (3)$$

Definition: Given a target maximum error $TargetErr$, a subfunction $g_i(x)$ is derivable from $g_j(x)$ if Eq. 4 is satisfied.

$$\exists T_i, Err(g_i, T_i, g_j) \leq TargetErr \quad (4)$$

Definition: The similarity matrix is defined as an $n \times n$ binary matrix, in which an entry sm_{ij} $(i \neq j)$ is 1 if and only if $g_i(x)$ is derivable from $g_j(x)$, subject to the constraint in Eq. 4.

$$SM = [sm_{ij}]_{n \times n};\ sm_{ij} \in \{0, 1\},$$
$$sm_{ij} = 1 \Leftrightarrow \exists T_i, Err(g_i, T_i, g_j) \leq TargetErr, i \neq j \quad (5)$$

Definition: The similarity vector is defined as a vector of n elements, equals to the summation of SM's rows.

$$SV = [sv_j]_{1 \times n};\ sv_j = \sum_i sm_{ij} \quad (6)$$

In HBU, each subfunction $g_i(x)$ is implemented using the PU method, and its initial bias b_i is stored in a binary lookup table. Then, a binary adder computes $f_i(x) = b_i + g_i(x)$. In our method, however, a small set of unique subfunctions are implemented, and each of the other subfunctions is derived from a transformation of one of the unique subfunctions. If $g_i(x)$ is derivable from $g_j(x)$, then we can conclude the following result from Eq. 4 and Eq. 1.

$$\begin{aligned} &\exists T_i, Err(g_i, T_i, g_j) \leq TargetErr \\ &\Rightarrow (b_i + g_i(x)) \simeq (b_i + bm_i) + T_i\{g_j(x)\} \\ &\Rightarrow f_i(x) \simeq \hat{b}_i + T_i\{g_j(x)\} \end{aligned}$$

If there are multiple transformations that can derive $g_i(x)$ from $g_j(f)$, we choose the one that minimizes the approximation error.

$$T_i = \arg\min_T Err(g_i, T, g_j) \quad (7)$$

Therefore, we can compute $f_i(x) \simeq \hat{b}_i + T_i\{g_j(x)\}$ instead of $f_i(x) = b_i + g_i(x)$. That is, we can implement $f_i(x)$ by storing the pre-calculated bias $\hat{b_i} = b_i + bm_i$ in the binary lookup table and transforming the subfunction $g_j(x)$, implemented using the PU method. Fig. 3 shows the architecture of our method.

As shown in Fig. 3, $g_i(x)$ is derived from $g_j(x)$ through the transformation T_i, and our method replaces the unary core $g_i(x)$ and its decoder with the simple transformer that might include NOT gates and/or shifters. It also reduces the fan-out of the input encoder. Such replacements make the hardware architecture less expensive compared to HBU, especially in non-monotonic functions at high resolutions.

To find the minimum set of unique subfunctions, we first compute the similarity matrix and similarity vector, as described in Eq. 5 and 6, respectively. The index of the maximum element in the similarity vector (i.e., $idx = \arg\max_i SV[i]$) determines the first unique function. Then, we traverse through the idx^{th} column of the similarity matrix to see which subfunctions can be derived from $g_{idx}(x)$. Next, we update the similarity matrix by zeroing the idx^{th} row and column as well as all the rows and columns corresponding to the functions that are derivable from $g_{idx}(x)$. Again, we compute the similarity vector and find the next unique subfunction. We continue this process until all the entries of the similarity matrix become zero. We do understand that dynamic programming could have been used to get a globally minimum number of unique subfunctions, but in this version, we have limited ourselves to this greedy approach, as it works very well in practice.

Algorithm 1 describes the procedure in more detail. To represent the final set of unique subfunctions in this algorithm, we define two vectors $Unique$ and $Transformer$ such that if $Unique[i] = j$ and $Transformer[i] = T_i$, then it means that $f_i(x)$ is derived from $f_j(x)$ through the transformation T_i. Vectors Sub and $Bias$ are also defined to store each subfunction's output values and bias.

Algorithm 1: SimBU Algorithm

Parameters: $TargetErr, w, w_b, w_u$
Input: $F = \{f(x)|x, f(x) \in \mathbb{Z} \text{ and } x, f(x) \in [0, 2^w)\}$
Outputs: $Sub, Bias, Unique, Transformer$
# Uniformly breaking the function into subfunctions
for $i = 1 \ \ to \ \ 2^{w_b}$ **do**
  $Sub[i] \leftarrow F[(i-1) \times 2^{w_u} : i \times 2^{w_u}]$
  $Bias[i] \leftarrow min(Sub[i])$
  $Sub[i] \leftarrow Sub[i] - Bias[i]$
end
# Making pairwise comparisons of the subfunctions
for $i = 1 \ \ to \ \ 2^{w_b}$ **do**
  for $j = 1 \ \ to \ \ 2^{w_b}$ **do**
    if $\exists T_i, Err(Sub[i], T_i, Sub[j]) \leq TargetErr$ **then**
      $SM[i][j] \leftarrow 1$
    else
      $SM[i][j] \leftarrow 0$
    end
  end
end
# Finding the minimum set of unique subfunctions
while $\sum_i \sum_j SM[i][j] \ != 0$ **do**
  $SV \leftarrow \sum_i SM[i][:]$
  $idx \leftarrow \arg\max_i SV[i]$
  for $i = 1 \ \ to \ \ 2^{w_b}$ **do**
    if $i \ != idx \ and \ SM[i][idx] == 1$ **then**
      $Unique[i] \leftarrow idx$
      $T_i \leftarrow \arg\min_T Err(Sub[i], T, Sub[idx])$
      $Transformer[i] \leftarrow T_i$
      $bm_i \leftarrow \lfloor \frac{1}{2^{w_u}} \sum (Sub[i] - T_i\{Sub[idx]\}) \rceil$
      $Bias[i] \leftarrow Bias[i] + bm_i$
      $SM[i][:] \leftarrow 0$
      $SM[:][i] \leftarrow 0$
    end
  end
  $Unique[idx] \leftarrow idx$
  $Transformer[idx] \leftarrow None$
  $SM[idx][:] \leftarrow 0$
  $SM[:][idx] \leftarrow 0$
end

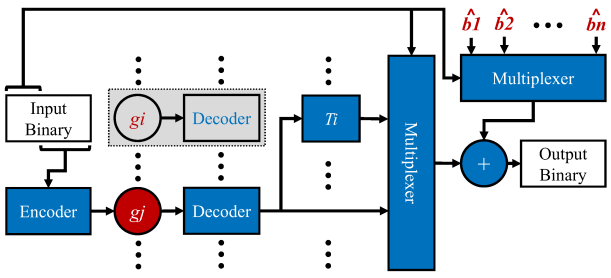


Fig. 3: Hardware architecture of our proposed method.

C. Guiding Example

In this section, we show how our method works for an arbitrary w-bit function $f(x) = [1 + sin(2\pi x)] \div 2$. We set the parameters as follows:

- w = 8 bits
- w_b = 2 bits $\Rightarrow w_u = w - w_b$ = 6 bits
- $TargetErr$ = 2^{-7}

We first divide the function into $2^{w_b} = 4$ subfunctions and separate their initial biases, as shown in Fig. 4a and 4b, respectively. The input range of all the subfunction is $w_u = 6$ bits.

$$f(x) = \begin{cases} f_1(x) = 128 + g_1(x) & x \in [0, 64) \\ f_2(x) = 131 + g_2(x) & x \in [64, 128) \\ f_3(x) = 0 + g_3(x) & x \in [128, 192) \\ f_4(x) = 0 + g_4(x) & x \in [192, 256) \end{cases} \quad (8)$$

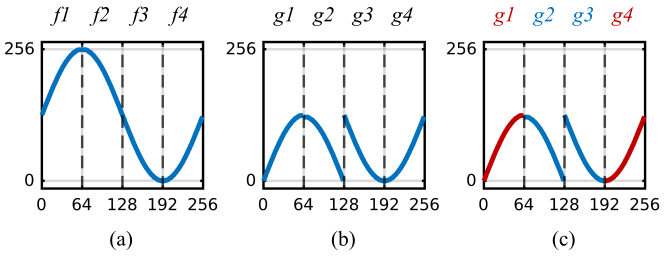


Fig. 4: subfunctions of the guiding example. The red subfunctions are unique subfunctions that derive the others.

Next, we make pairwise comparisons to find all the subfunctions that can be derived from each other. We can use the term *generate* as the inverse of deriving a function: if g_i can be derived from g_j through our linear transformations, then we say g_j generates g_i. Using Eq. 4, we find out that $g_1(x)$ can generate $g_3(x)$, $g_3(x)$ can generate $g_1(x)$, and $g_4(x)$ can generate $g_2(x)$.

$$\begin{aligned} &T_3 = inv \Rightarrow Err(g_3, inv, g_1) = 0.0038 \leq 2^{-7}, bm_3 = 1 \\ &T_1 = inv \Rightarrow Err(g_1, inv, g_3) = 0.0058 \leq 2^{-7}, bm_1 = -128 \\ &T_2 = inv \Rightarrow Err(g_2, inv, g_4) = 0.0058 \leq 2^{-7}, bm_2 = -3 \end{aligned} \quad (9)$$

Although $g_4(x)$ can generate $g_2(x)$ through an inverter, the opposite is not true. Due to the constrains on bm_4, as described in Eq. 3, we cannot find a transformation T_4 such that $Err(g_4, T_4, g_2) \leq TargetError = 2^{-7}$. For this reason, similarity matrices are not necessarily symmetric.

After finding the derivable subfunctions, we compute the similarity matrix SM and similarity vector SV, as shown in Fig. 5a. The index of the maximum entry in SV determines the first unique subfunction. In our example, the first, the third, and the fourth entries are the same, and choosing either one would be OK. As shown in Fig. 5b, we choose $f_1(x)$ as the first unique subfunction. Then, by traversing through the 1st column of SM, we can see $SM[3][1] = 1$. As a result:

$$\begin{aligned} SM[3][1] = 1 &\Rightarrow Err(g_3, inv, g_1) = 0.0038 \leq 2^{-7} \\ &\Rightarrow (0 + g_3(x)) \simeq (0 + 1) + inv\{g_1(x)\} \\ &\Rightarrow f_3(x) \simeq 1 + inv\{g_1(x)\} \end{aligned}$$

Therefore, we can implement $f_3(x) \simeq 1 + inv\{g_1(x)\}$ instead of $f_3(x) = 0 + g_3(x)$. Next, we update SM by zeroing the 1st and the 3rd rows and columns, and then recalculate SV, as shown in Fig. 5c. Again, Fig. 5d indicates that $g_4(x)$ is the next unique subfunction to be implemented. Similarly, we traverse through the 1st column and see $SM[2][4] = 1$. As a result:

$$\begin{aligned} SM[2][4] = 1 &\Rightarrow Err(g_2, inv, g_1) = 0.0058 \leq 2^{-7} \\ &\Rightarrow (131 + g_2(x)) \simeq (131 - 3) + inv\{g_4(x)\} \\ &\Rightarrow f_2(x) \simeq 128 + inv\{g_4(x)\} \end{aligned}$$

Therefore, $f_2(x) = 131 + g_2(x)$ converts to $f_2(x) \simeq 128 + inv\{g_4(x)\}$. As a reminder, the bias value 131 was the original b_2 from Eq. 8, and the value -3 was the bm_i from the transformation deriving $g_2(x)$ from $g_4(x)$ shown in Eq. 9. As shown in Fig. 5e, zeroing the 4th and 2nd rows and columns makes all the entries of SM get zero and ends the process.

As a result of our proposed algorithm, function $f(x)$ converts to the following function.

$$f(x) \simeq \hat{f}(x) = \begin{cases} 128 + g_1(x) & x \in [0, 64) \\ 128 + inv\{g_4(x)\} & x \in [64, 128) \\ 1 + inv\{g_1(x)\} & x \in [128, 192) \\ 0 + g_4(x) & x \in [192, 256) \end{cases}$$

Therefore, the number of subfunctions (including their scaling networks and decoders) reduces by half, and they are replaced by simple NOT gates. Fig. 4c shows that $g_1(x)$ and $f_4(x)$ are the unique subfunctions and $g_2(x)$ and $f_3(x)$ are derived from them.

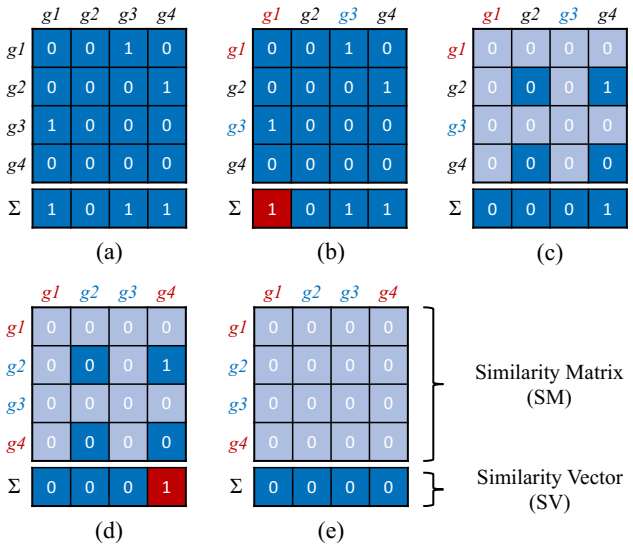


Fig. 5: Similarity matrix (SM) and similarity vector (SV) of the guiding example.

III. Methodology for Multivariate Functions

The unary methods can be extended to multivariate functions. A rudimentary idea with limited practical applications for extending the PU method to such functions was proposed in [11], [12]. In this paper, we only discuss 2-dimensional functions $f(x, y)$ for simplicity, although the approach can be applied to higher dimensions as well.

In PU for univariate functions, as discussed in Section II-A, an input unary value is mapped to an output unary value through the unary core. For a univariate function, this network consists of wires and XOR gates, as shown in Figure 1. However, the same does not hold for multivariate functions. In such functions, the method of [11], [12] finds a list of triggering points that dictate when the function value $f(x, y) \in \{0, 1, ..., 2^{w_{out}} - 1\}$ changes. For a certain output value $f = \alpha$, triggering points are those (x, y) pairs at which the function reaches or exceeds $f = \alpha$, *i.e.*, F_α is set to 1 for the first time[2]. For instance, $f(x, y) = xy$ reaches $f = 4$ at $(x, y) \in \{(1, 4), (2, 2), (4, 1)\}$. Therefore, F_4 must be triggered by:

$$F_4 = (X_1 \wedge Y_4) \vee (X_2 \wedge Y_2) \vee (X_4 \wedge Y_1)$$

In case of non-monotonically increasing functions, a list of alternating points must be detected besides triggering points. Alternating points are those inputs that alternate an output bit after it is triggered. Fig. 6 shows the output values of an arbitrary function along with its triggering and alternating points. In this function, the output reaches $f = 1$ for the first

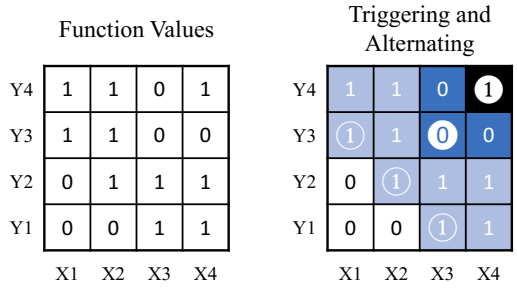


Fig. 6: The output values of an arbitrary function along with its triggering (white rings) and alternating (white circles) points.

time at $(x, y) \in \{(1, 3), (2, 2), (3, 1)\}$. Hence, these 3 tuples are considered as the triggering points for F_1. This output bit, however, alternates at $(x, y) \in \{(3, 3), (4, 4)\}$. As a result, the output bit F_1 must be triggered and alternated by:

$$F_1 = [(X_1 \wedge Y_3) \vee (X_2 \wedge Y_2) \vee (X_3 \wedge Y_1)] \oplus (X_3, Y_3) \oplus (X_4, Y_4)$$

Our method proposed in Section II-B can be applied to multivariate functions. Similar to univariate functions, we uniformly break the inputs of a multivariate function and separate the initial biases of the resulting subfunctions, however, the input segments in 2-D functions are square-shaped. Then, we measure the similarities between the subfunctions and find the minimum set of them that can reconstruct the original function using transformations given a target maximum error.

Compared to univariate functions, our method can have a greater impact on hardware cost reductions in multivariate functions. This is because multivariate functions require many AND and OR gates besides wires and XOR gates in the method of [11], [12]. As a result, using self-similarities and reducing the number of subfunctions can dramatically reduce the hardware cost.

[2] F_α is the α^{th} wire in bundle of 2^w wires in the unary domain.

TABLE I: Equations of the implemented univariate nonlinear functions.

Name	Equation
Log	$log2(x+1)$
Exp	$exp(x-1)$
Sigmoid	$1+tanh(4\times(2x-1))$
GELU	$0.5\times(6x-3)\times(erf(\frac{6x-3}{\sqrt{2}})+1)+0.25$
Sin	$1+sin(2\pi x)$
Sqr	x^2

IV. Implementation Results

A. Univariate Functions

We developed a Matlab script to run our proposed algorithm and generate Verilog files. Since our method optimizes the hardware cost given a target maximum error, we used two different target values: 2^{-w-1} and 2^{-w}. The former is equal to the maximum error of exact implementations, and therefore has the functions implemented with no approximation, whereas the latter is equal to the maximum error of approximating the least significant bit. These two types of implementations are denoted as SimBU-Exact and SimBU-Approx, respectively.

- SimBU-Exact: $\Leftrightarrow TargetError = 2^{-w-1}$
- SimBU-Approx $\Leftrightarrow TargetError = 2^{-w}$

Our C++ implementation of the SimBU-Exact algorithm generally takes about a fraction of a second to 1.5 seconds on a regular laptop computer. Given that the main run time bottleneck is the nested loop in lines 11-19 of Algorithm 1, using multi-threading or GPU acceleration can result in significant reduction in run time.

We evaluated our method and compared it to previous works including FloPoCo-LUT [10], HBU [18], and FloPoCo-PPA [3], [10]. The comparisons were made on the implementation of some functions at w = 8-, 10-, and 12-bit resolutions, and all the designs were synthesized on Xilinx's Kintex-7 FPGA using Vivado 2020.2 default design flow. Table I shows the equations of the implemented functions. The functions were evaluated on the unit interval $x \in$ [0, 1), and the outputs were scaled such that they range in the unit interval $f \in$ [0, 1). As a result, the fixed-point representations of the inputs and outputs consist of w fractional bits with no integer parts. Fig. 7 shows the graph of the implemented functions. Note that the decision to limit the range to [0, 1) is arbitrary. The input / output range could be anything, as long as the data is quantized and the minimum and maximum values are interpreted, *i.e.*, mapped to $0 \cdots 2^w$.

The FloPoCo-PPA cores were generated by the FixFunctionByPiecewisePoly tool in the FloPoCo framework. This tool generates VHDL files and evaluates a given function on [0, 1) using a piecewise polynomial approximation with the Horner scheme.

Table II shows the hardware cost and accuracy of each implemented function using FloPoCo-LUT, HBU, SimBU-Exact, FloPoCo-PPA, and SimBU-Approx methods. All results include the cost of unary encoders and decoders if the method uses unary encoding. To make fair comparisons between different methods in terms of area, we forced the synthesizer not to use any DSP and BRAM blocks, and the area was measured as the number of LUTs. In the tables, "Area" and "Delay" correspond to the number of LUTs and the critical path delay in nanoseconds, and "A × D" denotes the area × delay hardware cost. The "MaxErr" and "MSE" columns show the maximum absolute error and mean square error compared to double-precision floating-point implementations. The mean square error is defined in Eq. 10

$$MSE = \frac{1}{2^w}\sum_x (\hat{f}(x) - f(x))^2 \tag{10}$$

Although our method was expected to only reduce the hardware cost when implementing functions using approximation, it turned out that it could also reduce the cost compared to the HBU method even when not using approximations. To do so, the $TargetError$ parameter must be set to 2^{-w-1}, which is equivalent to the fixed-point quantization error. Since no approximation—other than the fixed-point quantization—is allowed, our proposed algorithm tries to find the same subfunctions after the basic bitwise transformations. That is, the subfunction g_i is considered derivable from g_j, only if there exists a transformation T_i such that the maximum error is equal to the minimum possible value (i.e., $\exists T_i, Err(g_i, T_i, g_j) \leq 2^{-w-1}$). This is a tough restriction on our algorithm. Surprisingly, the results in Table II show that SimBU-Exact outperforms HBU, especially at higher resolutions. This is because there are a large number of subfunctions at higher resolutions, and our method can reduce the number of unique subfunctions significantly, which compensates the added hardware resource for implementing the transformations T_i. It can be seen from the table that SimBU-Exact reduces the area × delay cost of FloPoCo-LUT and HBU on average by 36% and 32%, respectively, when averaged over 8-, 10- and 12-bit resolutions.

Table III also shows the number of subfunctions before and after the self-similarity measures using our proposed method. The total number of subfunctions in each function at each resolution depends on the parameters w_b and w_u, which are obtained experimentally. w_b and w_u are not independent, and the equation $w_u = w - w_b$ governs their relationship. Our Matlab script generates and synthesizes the Verilog files for different values of w_b to find the best value that minimizes the area × delay hardware cost.

Some applications—*e.g.*, machine learning and computer vision—can tolerate computational error to some extent, and computations can be performed approximately. In such cases, our method can be deployed well to implement arithmetic hardware cores. As the results in Table II show, SimBU-Approx improves the area × delay cost of FloPoCo-LUT, HBU, and FloPoCo-PPA on average by 60%, 56%, and 35%, respectively, again, when averaged over 8-, 10-, and 12-bit resolutions. As seen in the table, our method fully utilizes the error budget to simplify the hardware architecture and reduce

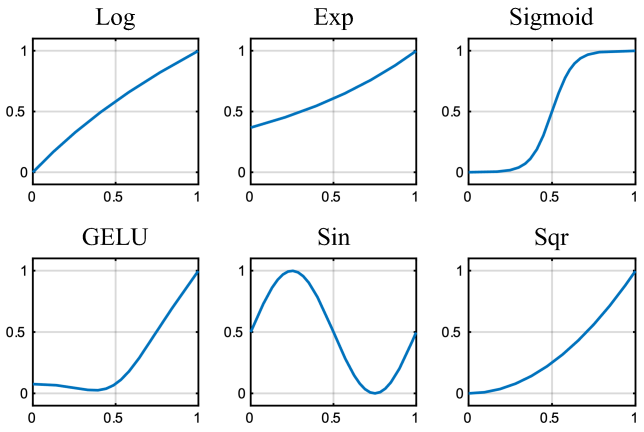


Fig. 7: Graphs of the implemented univariate nonlinear functions.

the cost further compared to FloPoCo-PPA, which was given the same error budget but could not fully utilize it. The gap between our A × D and that of FloPoCo-PPA gets smaller as the bit width increases. FloPoCo-PPA is well-known to be good at higher resolutions, and less so on lower resolutions, and our results show that too.

The authors in [18] show that HBU performs better than the conventional binary, PU [12], and SC methods [13]–[16] at 8-, 10-, and 12-bit resolutions. Therefore, we can conclude that our method also outperforms all these previous works at 8-, 10-, and 12-bit resolutions. It is worth noting that piecewise polynomial approximation methods (e.g., FloPoCo-PPA) are used to reduce the size and complexity of a function's lookup table by approximating the pieces of the function with polynomials. Yet, these methods eventually need to store the coefficients of the polynomials in lookup tables. As our results show, SimBU-Exact can efficiently implement the functions without accuracy loss. Therefore, one might combine a PPA method and SimBU-Exact to implement a function at higher resolutions more efficiently. For instance, FloPoCo-PPA can be used to reduce the complexity of a 32-bit function's lookup table, and then SimBU-Exact might be used to implement the resulting lookup tables at lower hardware cost with no further approximation error.

B. Multivariate Functions

We evaluated our SimBU-Exact and SimBU-Approx methods and compared them to HBU [18] for multivariate functions by implementing a number of 2-dimensional functions at w = 8-bit resolution. All the designs were synthesized on Xilinx's Kintex-7 FPGA using Vivado 2020.2 default design flow. Table IV and Fig. 8 show the equations and graphs of the implemented multivariate nonlinear functions, respectively. The functions were evaluated on the unit interval $x \in [0, 1)$, and the outputs were scaled such that they range in the unit interval $f \in [0, 1)$.

The idea of implementing multivariate functions using PU was proposed in [11], [12]. However, other unary methods can be extended to multivariate functions as well. Although the authors in [17], [18] did not provide the results of implementation of any multivariate functions using HBU, we implemented the multivariate functions shown in Table IV using both PU and HBU methods, besides SimBU-Exact and SimBU-Approx. Since the area × delay hardware cost of HBU was by far better than PU, we only compared our method to HBU.

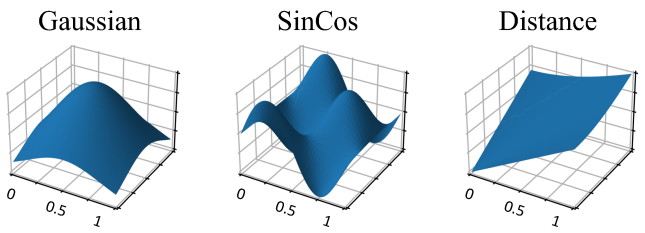


Fig. 8: Graphs of the implemented multivariate nonlinear functions.

Table V shows the hardware cost and accuracy of each implemented function using HBU, SimBU-Exact, and SimBU-Approx methods. The table shows that SimBU-Exact improves the area × delay hardware cost of HBU on average by 78% with no approximations and by 92% given the approximation of the least significant bit. The improvement of our SimBU method over HBU for multivariate functions is greater than univariate functions, because unary methods in multivariate functions require many AND and OR gates besides wires and XOR gates, and our method can reduce the number of such gates by reducing the number of subfunctions. Table VI shows the number of subfunctions before and after the self-similarity measures using our proposed method.

V. Application

Nonlinear functions are frequently used in many applications such as machine learning and image processing. For instance, Transformers [36], which are deep learning models, heavily use nonlinear functions GELU, Softmax, and LayerNorm. Many works have been proposed to accelerate such nonlinear operations in BERT (bidirectional encoder representations from transformers) as a transformer [19], [37]–[39]. In our previous work [19], we demonstrated how an approximate HBU method can benefit nonlinear operations in BERT. The sometimes large maximum absolute errors introduced by that work might not be an issue in machine learning applications that possess intrinsic error tolerance characteristics [40]–[42]. However, maximum absolute error can directly impact accuracy in many applications that are more sensitive to approximations.

In this section, we evaluate our method on two different image processing applications: homomorphic filtering and Robert's cross edge detection. All designs were synthesized on Xilinx's Kintex-7 FPGA using Vivado 2020.2 default design flow.

A. Homomorphic Filtering

In PET or CT scans, the ability to find hot spots without them obscuring nearby details is of vital importance. Unfortunately, hot spots usually add multiplicative noise to an image,

TABLE II: Univariate nonlinear functions' hardware cost and accuracy results. Note that the maximum error in the exact methods is constant because that is the quantization error based on the bit resolution. In the approximate methods, the maximum error is bounded by design.

Exact Methods

FloPoCo-LUT [10]	**8-bit**					**10-bit**					**12-bit**				
	Area	**Delay**	**A × D**	**MaxErr**	**MSE**	**Area**	**Delay**	**A × D**	**MaxErr**	**MSE**	**Area**	**Delay**	**A × D**	**MaxErr**	**MSE**
Log	29	1.40	40.54	1.94E-03	1.38E-06	137	2.03	278.38	4.88E-04	8.35E-08	508	2.49	1,262.89	1.22E-04	4.89E-09
Exp	29	1.53	44.43	1.95E-03	1.23E-06	133	2.02	268.79	4.88E-04	7.95E-08	468	2.43	1,136.30	1.22E-04	4.94E-09
Sigmoid	24	1.39	33.36	1.95E-03	1.28E-06	97	2.01	194.97	4.88E-04	8.24E-08	379	2.48	939.16	1.22E-04	4.99E-09
GELU	25	1.37	34.13	1.95E-03	1.46E-06	114	1.97	224.35	4.88E-04	7.87E-08	462	2.41	1,115.27	1.22E-04	5.07E-09
Sin	29	1.40	40.51	1.95E-03	1.44E-06	126	2.02	254.27	4.88E-04	7.36E-08	533	2.47	1,318.11	1.22E-04	4.92E-09
Sqr	27	1.53	41.34	1.95E-03	1.22E-06	135	2.02	272.70	4.78E-04	7.74E-08	574	2.50	1,433.85	1.22E-04	4.94E-09
Average			**39.05**					**248.91**					**1,200.93**		

HBU [18]	**8-bit**					**10-bit**					**12-bit**				
	Area	**Delay**	**A × D**	**MaxErr**	**MSE**	**Area**	**Delay**	**A × D**	**MaxErr**	**MSE**	**Area**	**Delay**	**A × D**	**MaxErr**	**MSE**
Log	31	1.54	47.59	1.94E-03	1.38E-06	72	2.84	204.19	4.88E-04	8.35E-08	318	3.31	1,053.22	1.22E-04	4.89E-09
Exp	28	1.53	42.92	1.95E-03	1.23E-06	64	2.83	181.31	4.88E-04	7.95E-08	279	3.38	943.02	1.22E-04	4.94E-09
Sigmoid	24	1.39	33.36	1.95E-03	1.28E-06	96	2.01	193.06	4.88E-04	8.24E-08	268	3.30	884.94	1.22E-04	4.99E-09
GELU	27	1.30	35.18	1.95E-03	1.46E-06	83	2.75	228.58	4.88E-04	7.87E-08	267	3.35	895.52	1.22E-04	5.07E-09
Sin	31	1.65	51.27	1.95E-03	1.44E-06	92	2.86	263.12	4.88E-04	7.36E-08	544	3.77	2,051.97	1.22E-04	4.92E-09
Sqr	28	1.24	34.72	1.95E-03	1.22E-06	68	2.71	184.42	4.78E-04	7.74E-08	300	3.39	1,017.30	1.22E-04	4.94E-09
Average			**40.84**					**209.11**					**1,140.99**		

SimBU-Exact (our method)	**8-bit**					**10-bit**					**12-bit**				
	Area	**Delay**	**A × D**	**MaxErr**	**MSE**	**Area**	**Delay**	**A × D**	**MaxErr**	**MSE**	**Area**	**Delay**	**A × D**	**MaxErr**	**MSE**
Log	20	2.34	46.70	1.94E-03	1.38E-06	70	2.51	175.56	4.88E-04	8.35E-08	227	3.02	685.31	1.22E-04	4.89E-09
Exp	18	2.21	39.73	1.95E-03	1.23E-06	62	2.52	156.24	4.88E-04	7.95E-08	239	2.78	663.70	1.22E-04	4.94E-09
Sigmoid	26	1.39	36.24	1.95E-03	1.28E-06	74	2.52	186.70	4.88E-04	8.24E-08	242	2.95	713.66	1.22E-04	4.99E-09
GELU	25	1.37	34.13	1.95E-03	1.46E-06	70	2.55	178.22	4.88E-04	7.87E-08	242	3.03	732.29	1.22E-04	5.07E-09
Sin	21	1.38	29.00	1.95E-03	1.44E-06	75	1.82	136.50	4.88E-04	7.36E-08	291	3.13	911.12	1.22E-04	4.92E-09
Sqr	18	2.27	40.91	1.95E-03	1.22E-06	69	2.52	174.02	4.78E-04	7.74E-08	254	2.90	736.35	1.22E-04	4.91E-09
Average			**37.79**					**167.87**					**740.41**		
Improvement Over FloPoCo-LUT			**3.24%**					**32.56%**					**38.35%**		
Improvement Over HBU			**7.48%**					**19.72%**					**35.11%**		

Approximate Methods

FloPoCo-PPA [3], [10]	**8-bit**					**10-bit**					**12-bit**				
	Area	**Delay**	**A × D**	**MaxErr**	**MSE**	**Area**	**Delay**	**A × D**	**MaxErr**	**MSE**	**Area**	**Delay**	**A × D**	**MaxErr**	**MSE**
Log	48	4.67	224.06	2.90E-03	1.62E-06	67	4.27	286.02	7.95E-04	1.04E-07	108	5.39	581.69	2.05E-04	6.01E-09
Exp	48	4.51	216.48	3.27E-03	1.82E-06	62	4.29	265.67	8.81E-04	1.35E-07	97	4.91	476.46	2.00E-04	6.73E-09
Sigmoid	41	3.25	133.29	3.53E-03	1.51E-06	61	4.33	264.31	8.56E-04	9.77E-08	116	5.08	589.74	2.37E-04	6.38E-09
GELU	41	3.32	136.20	2.65E-03	1.88E-06	59	4.31	254.00	7.93E-04	1.12E-07	104	5.28	548.91	2.05E-04	7.07E-09
Sin	39	3.32	129.36	2.77E-03	1.95E-06	60	4.39	263.16	8.78E-04	1.11E-07	101	5.21	525.71	2.28E-04	7.10E-09
Sqr	26	3.53	91.70	3.65E-03	2.58E-06	48	4.11	197.47	8.68E-04	1.25E-07	63	3.94	247.91	2.39E-04	9.25E-09
Average			**155.18**					**255.11**					**495.07**		

SimBU-Approx (our method)	**8-bit**					**10-bit**					**12-bit**				
	Area	**Delay**	**A × D**	**MaxErr**	**MSE**	**Area**	**Delay**	**A × D**	**MaxErr**	**MSE**	**Area**	**Delay**	**A × D**	**MaxErr**	**MSE**
Log	19	1.94	36.86	3.88E-03	3.05E-06	28	2.21	61.99	9.77E-04	1.67E-07	122	3.25	396.50	2.42E-04	9.27E-09
Exp	16	2.20	35.15	3.79E-03	2.60E-06	25	2.21	55.28	9.75E-04	1.69E-07	94	3.64	341.78	2.43E-04	1.05E-08
Sigmoid	14	1.98	27.69	3.86E-03	1.78E-06	52	2.59	134.78	9.67E-04	1.26E-07	155	3.48	539.87	2.44E-04	8.15E-09
GELU	14	1.98	27.66	3.75E-03	1.89E-06	41	2.45	100.41	9.76E-04	1.25E-07	142	2.81	399.02	2.44E-04	7.87E-09
Sin	21	1.38	29.00	1.95E-03	1.44E-06	75	1.82	136.50	4.88E-04	7.36E-08	164	3.51	576.13	2.44E-04	1.10E-08
Sqr	17	2.19	37.30	3.87E-03	2.41E-06	53	2.81	149.09	9.76E-04	1.72E-07	125	3.71	464.25	2.44E-04	1.03E-08
Average			**32.28**					**106.34**					**452.93**		
Improvement Over FloPoCo-LUT			**17.34%**					**57.28%**					**62.29%**		
Improvement Over HBU			**20.97%**					**49.15%**					**60.30%**		
Improvement Over FloPoCo-PPA			**79.20%**					**58.31%**					**8.51%**		

TABLE III: Number of subfunctions before (Total) and after (Unique) the proposed self-similarity measures in the univariate nonlinear functions.

SimBU-Exact	8-bit		10-bit		12-bit	
	Total	Unique	Total	Unique	Total	Unique
Log	64	5	128	35	512	39
Exp	64	5	256	5	1024	5
Sigmoid	2	2	256	16	1024	16
GELU	2	2	256	11	512	53
Sin	4	2	4	2	128	64
Sqr	64	7	256	8	1024	8

SimBU-Approx	8-bit		10-bit		12-bit	
	Total	Unique	Total	Unique	Total	Unique
Log	32	3	64	7	128	16
Exp	64	1	64	5	256	7
Sigmoid	64	2	256	4	512	9
GELU	64	1	128	3	512	4
Sin	4	2	4	2	512	7
Sqr	64	1	128	5	256	11

TABLE IV: Equations of the implemented multivariate nonlinear functions.

Name	Equation
Gaussian	$\frac{1}{2\pi\sqrt{\det\Sigma}} exp(-\frac{1}{2}(\begin{bmatrix} x \\ y \end{bmatrix} - \mu)^T \Sigma^{-1} (\begin{bmatrix} x \\ y \end{bmatrix} - \mu))$
SinCos	$1 + sin(2\pi x) \times cos(2\pi y)$
Distance	$\sqrt{x^2 + y^2}$

making it harder to distinguish other nearby details. Due to it being multiplicative, applying simple linear filters on these images would not be sufficient for removing the noise. An example of such an image is shown in Fig. 11a, where the full-body PET scan shows two hot spots. The primary issue is that these hot spots dominate the dynamic range, making other features near the brain and lung dimmer and blurrier. Homomorphic filtering was designed to fix such issues by filtering out the multiplicative noise while also rectifying the dynamic range. Homomorphic filtering also has applications in neurocomputing to decode information from the spiking sequence of a neuron model [43], but for this work we will explore homomorphic filtering as an image enhancement technique.

Assuming an illumination-reflectance model [44], an image is decomposed as a product of the illumination $i(x, y)$ and reflectance $r(x, y)$.

$$f(x, y) = i(x, y) \times r(x, y)$$

The illumination is the key feature that we want to preserve (*e.g.*, hot spots and edges of organs), while reflectance has the high frequency multiplicative noise we want to remove. Due to both illumination and reflectance being combined multiplicatively, a linear filter cannot be applied directly to $f(x, y)$ without removing some information in illumination.

TABLE V: Multivariate nonlinear functions' hardware cost and accuracy results.

Exact Methods

HBU [18]	8-bit				
	Area	Delay	A × D	MaxErr	MSE
Gaussian	18120	7.03	127,347.36	1.95E-03	1.28E-06
SinCos	8462	7.14	60,444.07	1.95E-03	1.29E-06
Distance	5354	6.31	33,783.74	1.95E-03	1.29E-06
Average			**73,858.39**		

SimBU-Exact (our method)	8-bit				
	Area	Delay	A × D	MaxErr	MSE
Gaussian	5500	5.02	27,632.00	1.95E-03	1.28E-06
SinCos	2372	4.44	10,529.31	1.95E-03	1.29E-06
Distance	2370	4.80	11,385.48	1.95E-03	1.29E-06
Average			**16,515.60**		
Improvement Over HBU			**77.64%**		

Approximate Method

SimBU-Approx (our method)	8-bit				
	Area	Delay	A × D	MaxErr	MSE
Gaussian	2123	3.86	8,190.53	3.91E-03	2.63E-06
SinCos	1996	3.74	7,471.03	3.90E-03	2.52E-06
Distance	336	4.53	1,522.08	3.91E-03	2.43E-06
Average			**5,727.88**		
Improvement Over HBU			**92.24%**		

TABLE VI: The number of subfunctions before (Total) and after (Unique) the proposed self-similarity measures in the multivariate nonlinear functions.

SimBU-Exact	8-bit	
	Total	Unique
Gaussian	4096	1393
SinCos	4096	697
Distance	4096	130

SimBU-Approx	8-bit	
	Total	Unique
Gaussian	4096	29
SinCos	4096	54
Distance	1024	14

This can be resolved by first applying a log-transformation, making the illumination and reflectance additive.

$$ln(f(x, y)) = ln(i(x, y)) + ln(r(x, y))$$

We can then remove noise by applying linear filters and then taking an exponential to get our cleaned final image, as shown in Fig. 11b. As seen in the figure, homomorphic filtering not only controlled the dynamic range between the hotspots and the rest of the image, but also applied noise removal to make the features within the PET scan more legible.

Typically, the filtering is done by applying an FFT to the log-transformed image, but for our FPGA application, we are doing it in the spatial domain by convolving a 5 × 5 high-pass filter kernel. This allows us to evaluate an end-to-end homomorphic filter which works on a 5 × 5 sized window that moves across the entire image. We deployed our method and previous works such as FloPoCo-LUT, HBU, and FloPoCo-PPA to replace the nonlinear functions described below to compare the area × delay hardware cost and accuracy. A block

diagram of this is shown in Fig. 9. The Log and Exp layers perform the following functions at 10-bit resolution:

- Log: $ln(x+1)$
- Exp: $(\frac{1-2^{-10}}{exp(1)}) \times exp(2x-1)$

To support FloPoCo-PPA [3], [10] and get good accuracy, the functions and convolution output are re-scaled and shifted such that the input and output ranges of each nonlinear function fit the range $[0, 1)$. Since each input and output scale is a predetermined constant, we do not require more operations to scale our quantized values, and these scaling constants are built into the nonlinear functions. Since the convolution outputs a signed fixed-point value, we add and shift the output to convert the range to $[0, 1)$ before passing through the Exp layer, in which $2x-1$ corrects for the range change. In the convolution, bit lengths are automatically extended to ensure no overflow in multiplication and accumulation. Assuming a Gaussian input, a predetermined shift amount was calculated to do a saturate shift on the accumulated output to bring the bit-length back down for the Exp layer.

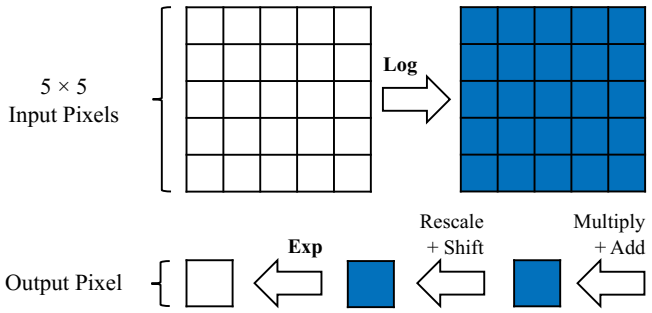


Fig. 9: Block diagram of the homomorphic filtering on a 5×5 window.

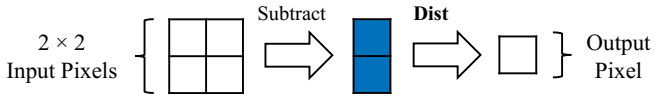
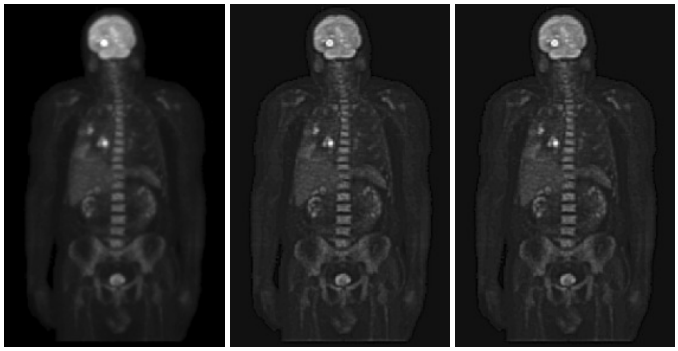


Fig. 10: Block diagram of the Robert's Cross edge detection.

able VII shows the hardware cost and accuracy of the implemented homomorphic filter. For accuracy results, we compared our final image against the reference result where the operations are being done in floating-point. Since the function in the Exp layer is only applied once to the output pixel, we implemented this layer using SimHBU-Exact for both variants of SimHBU. As seen in the table, our SimBU-Exact method reduces the area × delay hardware cost of HBU by 7% with the same quality, while our SimBU-Approx method reduces the hardware cost by 44% with a 0.3% quality loss. Compared to FloPoCo-PPA, our SimBU-Approx method reduces the hardware cost by 46% with higher quality. The final image with SimBU-Approx is shown in Fig. 11c, and it is nearly equivalent to the reference image.



(a) Input (b) Floating Point (c) Our method

Fig. 11: Test images of a PET scan in the homomorphic filtering. Input image sourced from [44]. Part (b) shows the homomorphic filtering applied to the input image using floating point computations. Part (c) shows the results from the homomorphic filtering using the SimBU-Approx method.

TABLE VII: Homomorphic filtering's hardware cost and accuracy results.

Exact Methods					
Method	**Area**	**Delay**	**A × D**	**MSE**	**PSNR**
FloPoCo-LUT [10]	4258	22.63	96,341.51	4.37E-01	51.72
HBU [18]	2738	24.03	65,796.88	4.37E-01	51.72
SimBU-Exact (our method)	2538	23.99	60,879.01	4.37E-01	51.72
Approximate Methods					
Method	**Area**	**Delay**	**A × D**	**MSE**	**PSNR**
FloPoCo-PPA [3], [10]	2430	27.83	67,629.33	4.62E-01	51.48
SimBU-Approx (our method)	1538	23.76	36,535.19	4.56E-01	51.54

B. Robert's Cross Edge Detection

Robert's cross [45] is a gradient-based operation performed to detect edges in an image as follows:

$$G(x,y) = \sqrt{{G_x}^2(x,y) + {G_y}^2(x,y)} \tag{11}$$

where $G_x(x,y)$ and $G_y(x,y)$ are given by:

$$G_x(x,y) = p(x,y) - p(x+1,y+1)$$
$$G_y(x,y) = p(x+1,y) - p(x,y+1)$$

where $p(x,y)$ denotes an input pixel. Although this simple operator can detect edges, it is highly sensitive to noise [45].

We deployed our method and the previous HBU work to implement the nonlinear function of the Robert's cross operation described in Fig. 10. The Dist layer performs the following function at 8-bit resolution:

- Dist: $\sqrt{x^2+y^2}$

It is worth noting that we could perform the Robert's cross operation by implementing a 4-dimensional nonlinear function or by breaking Eq. 11 and implementing three 1-dimensional nonlinear functions, which might result in better hardware cost. However, here we implemented it using a 2-dimensional nonlinear function to evaluate the trade-off between accuracy and hardware cost provided by our method for multivariate functions in such an error-sensitive application. In general,

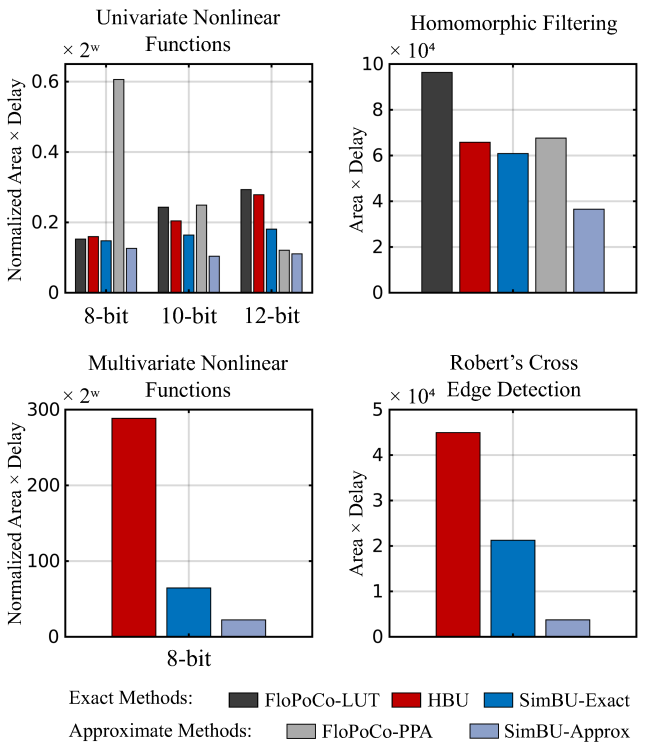


Fig. 12: Comparison of hardware cost in the implemented nonlinear functions and applications.

TABLE VIII: Robert's cross edge detection's hardware cost and accuracy results.

Exact Methods					
Method	**Area**	**Delay**	**A × D**	**MSE**	**PSNR**
HBU [18]	5416	8.30	44,947.38	6.98E-06	51.56
SimBU-Exact (our method)	2483	8.56	21,247.03	6.98E-06	51.56
Approximate Method					
Method	**Area**	**Delay**	**A × D**	**MSE**	**PSNR**
SimBU-Approx (our method)	546	6.84	3,733.00	7.53E-06	51.23

we only claim our method to be effective when dealing with univariate functions for sure, and to a certain extent, 2-dimensional functions. Going beyond two variables might work for certain functions, but it has the risk of exponentially large solution space.

Table VIII shows the hardware cost and the accuracy of the implemented Robert's cross edge detection. We can see that our SimBU-Exact method improves the area × delay hardware cost of HBU by 53% with the same quality, and our SimBU-Approx method improves that by 92% with a 0.6% quality loss. Fig. 10 shows the input and output images generated using floating-point computations and our SimBU-Approx method. Finally, Fig. 12 compares the area × delay hardware cost of the implemented nonlinear functions and applications.

VI. Conclusions

In this work, we proposed a method to implement nonlinear functions given a target maximum absolute error. It provides a trade-off between accuracy and hardware cost by reducing the number of subfunctions in the previous HBU (hybrid binary-unary) method and replacing them with simple bitwise transformers. In terms of area × delay hardware cost, our results show that our method outperforms the FloPoCo-LUT (lookup table) and HBU methods with no approximation error, and by approximating the least significant bit and using the same error budget, it outperforms the FloPoCo-PPA (piecewise polynomial approximation) method at up to 12-bit resolutions as well. Finally, we implemented homomorphic filtering and Robert's cross edge detection as image processing applications to show the benefits of our method compared to previous works. Without loss of quality, our method implemented both applications at lower hardware cost than the previous exact and approximate methods.

(a) Input (b) Floating Point (c) Our method

Fig. 13: Test images in the Robert's cross edge detection. Part (b) shows the Robert's cross edge detection applied to the input image using floating point computations. Part (c) shows the results from the Robert's cross edge detection using the SimBU-Approx method.

Acknowledgments

This material is based upon work supported in part by Cisco Systems, Inc. under grant number 1085913, and by the National Science Foundation under grant number PFI-TT 2016390.

References

[1] M. H. Najafi, D. J. Lilja, M. D. Riedel, and K. Bazargan, "Low-cost sorting network circuits using unary processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 8, pp. 1471–1480, 2018.

[2] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, ser. FPGA '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 191–200. [Online]. Available: https://doi.org/10.1145/275107.275139

[3] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," in *2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*, 2005, pp. 328–333.

[4] H. Dong, M. Wang, Y. Luo, M. Zheng, M. An, Y. Ha, and H. Pan, "Plac: Piecewise linear approximation computation for all nonlinear unary functions," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 9, pp. 2014–2027, 2020.

[5] Y. Tian, T. Wang, Q. Zhang, and Q. Xu, "Approxlut: A novel approximate lookup table-based accelerator," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 438–443.

[6] J. T. Butler, C. Frenzen, N. Macaria, and T. Sasao, "A fast segmentation algorithm for piecewise polynomial numeric function generators," *Journal of Computational and Applied Mathematics*, vol. 235, no. 14, pp. 4076–4082, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S037704271100121X

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

[8] B. Adcock, S. Brugiapaglia, and C. G. Webster, "Compressed sensing approaches for polynomial approximation of high-dimensional functions," in *Compressed Sensing and Its Applications: Second International MATHEON Conference 2015*. Springer, 2017, pp. 93–124.

[9] C. Pradhan, M. Letras, and J. Teich, "Efficient table-based function approximation on fpgas using interval splitting and bram instantiation," *ACM Trans. Embed. Comput. Syst.*, jan 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3580737

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

[11] S. Mohajer, Z. Wang, and K. Bazargan, "Routing magic: Performing computations using routing networks and voting logic on unary encoded data," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 77–86. [Online]. Available: https://doi.org/10.1145/3174243.3174267

[12] S. Mohajer, Z. Wang, K. Bazargan, and Y. Li, "Parallel unary computing based on function derivatives," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 14, no. 1, oct 2020. [Online]. Available: https://doi.org/10.1145/3418464

[13] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 93–105, 2011.

[14] Z. Wang, N. Saraf, K. Bazargan, and A. Scheel, "Randomness meets feedback: Stochastic implementation of logistic map dynamical system," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2744769.2744898

[15] S. A. Salehi, Y. Liu, M. D. Riedel, and K. K. Parhi, "Computing polynomials with positive coefficients using stochastic logic by double-nand expansion," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, ser. GLSVLSI '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 471–474. [Online]. Available: https://doi.org/10.1145/3060403.3060410

[16] P. Li, D. J. Lilja, W. Qian, M. D. Riedel, and K. Bazargan, "Logical computation on stochastic bit streams with linear finite-state machines," *IEEE Transactions on Computers*, vol. 63, no. 6, pp. 1474–1486, 2014.

[17] S. R. Faraji and K. Bazargan, "Hybrid binary-unary hardware accelerator," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 210–215. [Online]. Available: https://doi.org/10.1145/3287624.3287706

[18] ——, "Hybrid binary-unary hardware accelerator," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1308–1319, 2020.

[19] A. Khataei, G. Singh, and K. Bazargan, "Approximate hybrid binary-unary computing with applications in bert language model and image processing," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 165–175. [Online]. Available: https://doi.org/10.1145/3543622.3573181

[20] ——, "Optimizing hybrid binary-unary hardware accelerators using self-similarity measures," in *2023 IEEE 31th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023.

[21] J.-M. Muller, "Elementary functions and approximate computing," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2136–2149, 2020.

[22] Y. Chen and H. Li, "Stochastic computing using amplitude and frequency encoding," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 5, pp. 656–660, 2022.

[23] M. H. Najafi, D. Jenson, D. J. Lilja, and M. D. Riedel, "Performing stochastic computation deterministically," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 12, pp. 2925–2938, 2019.

[24] A. Morán, L. Parrilla, M. Roca, J. Font-Rossello, E. Isern, and V. Canals, "Digital implementation of radial basis function neural networks based on stochastic computing," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 13, no. 1, pp. 257–269, 2023.

[25] A. Alaghi, W. Qian, and J. P. Hayes, "The promise and challenge of stochastic computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 8, pp. 1515–1531, 2018.

[26] K. Chen, Y. Gao, H. Waris, W. Liu, and F. Lombardi, "Approximate softmax functions for energy-efficient deep neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 1, pp. 4–16, 2023.

[27] Y. Zhang, J. Qin, J. Han, and G. Xie, "Design of a stochastic computing architecture for the phansalkar algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 32, no. 3, pp. 442–454, 2024.

[28] X. Wei and L. Xiu, "A vlsi digital circuit platform for performing deterministic stochastic computing in the time dimension using fraction operations on rational numbers," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 1, pp. 194–207, 2023.

[29] J. Wang, H. Chen, D. Wang, K. Mei, S. Zhang, and X. Fan, "A noise-driven heterogeneous stochastic computing multiplier for heuristic precision improvement in energy-efficient dnns," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 2, pp. 630–643, 2023.

[30] H. Guo, Y. Zhao, Z. Li, Y. Hao, C. Liu, X. Song, X. Li, Z. Du, R. Zhang, Q. Guo, T. Chen, and Z. Xu, "Cambricon-u: A systolic random increment memory architecture for unary computing," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 424–437. [Online]. Available: https://doi.org/10.1145/3613424.3614286

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

[32] F. de Dinechin and A. Tisserand, "Multipartite table methods," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 319–330, 2005.

[33] S.-F. Hsiao, C.-S. Wen, Y.-H. Chen, and K.-C. Huang, "Hierarchical multipartite function evaluation," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 89–99, 2017.

[34] 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," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 5, pp. 466–470, 2015.

[35] 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.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[37] J. Yu, J. Park, S. Park, M. Kim, S. Lee, D. H. Lee, and J. Choi, "Nn-lut: Neural approximation of non-linear operations for efficient transformer inference," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 577–582. [Online]. Available: https://doi.org/10.1145/3489517.3530505

[38] H. Khan, A. Khan, Z. Khan, L. B. Huang, K. Wang, and L. He, "Npe: an fpga-based overlay processor for natural language processing," *arXiv preprint arXiv:2104.06535*, 2021.

[39] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-bert: Integer-only bert quantization," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 5506–5518. [Online]. Available: https://proceedings.mlr.press/v139/kim21d.html

[40] M. S. Ansari, V. Mrazek, B. F. Cockburn, L. Sekanina, Z. Vasicek, and J. Han, "Improving the accuracy and hardware efficiency of neural networks using approximate multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 317–328, 2020.

[41] T. Na and S. Mukhopadhyay, "Speeding up convolutional neural network training with dynamic precision scaling and flexible multiplier-accumulator," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ser. ISLPED '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 58–63. [Online]. Available: https://doi.org/10.1145/2934583.2934625

[42] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: Energy-efficient neuromorphic systems using approximate computing," in *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ser. ISLPED '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 27–32. [Online]. Available: https://doi.org/10.1145/2627369.2627613

[43] S. Orcioni, A. Paffi, F. Camera, F. Apollonio, and M. Liberti, "Automatic decoding of input sinusoidal signal in a neuron model: High pass homomorphic filtering," *Neurocomputing*, vol. 292, pp. 165–173, 05 2018.

[44] R. Gonzalez and R. Woods, *Digital Image Processing*. Pearson, 2018. [Online]. Available: https://books.google.com/books?id=0F05vgAACAAJ

[45] L. S. Davis, "A survey of edge detection techniques," *Computer graphics and image processing*, vol. 4, no. 3, pp. 248–270, 1975.

Alireza Khataei received the B.Sc. degree in electrical engineering from Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran, in 2021. He is currently a Ph.D. student at the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN, USA. He was a nominee of the Best Paper Award at ISFPGA in 2024. His current research focuses on hardware accelerator design for compute-intensive applications using approximate computing.

Gaurav Singh received his Bachelors degree in computer engineering from University of Minnesota, Minneapolis, MN, USA, in 2018. He is currently a Ph.D. candidate under Prof. Kia Bazargan at the Department of Electrical and Computer Engineering, University of Minnesota. His current research is in hardware-software co-design for neural network compression, with focus on using approximate and unary computing hardware implementations.

Kia Bazargan received the B.Sc. degree in Computer Science from Sharif University, Tehran, Iran, and the M.S. and Ph.D. degrees in Electrical and Computer Engineering from Northwestern University, Evanston, IL, USA, in 1998 and 2000, respectively. He is currently the Leroy and Ruth Fingerson Co-op Professor and the Director of the co-op program at the College of Science and Egnineering, and an Associate Professor with the Department of Electrical and Computer Engineering at the University of Minnesota, Minneapolis, MN, USA. He was a recipient of the U.S. National Science Foundation Career Award in 2004. He has been on the Technical Program Committee of all four major FPGA conferences (ISFPGA, FCCM, FPL, and FPT), as well as number of other IEEE/ACM sponsored conferences, including ICCAD, DAC, and ICCD.