2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) | 979-8-3503-1205-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/FCCM57271.2023.00020

Optimizing Hybrid Binary-Unary Hardware Accelerators Using Self-Similarity Measures

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 non-linear functions using few hardware resources compared to binary computing. 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 method and provide it with the trade-off between accuracy and hardware cost by introducing controlled levels of approximation. Given a target maximum error, our method breaks a function into sub-functions and tries to find the minimum set of unique sub-functions that can derive all the other ones through trivial bit-wise transformations. We compare our method to previous works such as HBU (hybrid binary-unary) and FloPoCo-PPA (piece-wise polynomial approximation) on a number of non-linear functions including Log, Exp, Sigmoid, GELU, Sin, and Sqr, which are used in neural networks and image processing applications. Without any loss of accuracy, our method can improve the area-delay-product hardware cost of HBU on average by 7% at 8-bit, 20% at 10-bit, and 35% at 12-bit resolutions. Given the approximation of the least significant bit, our method reduces the hardware cost of HBU on average by 21% at 8-bit, 49% at 10-bit, and 60% at 12-bit resolutions, and using the same error budget as given to FloPoCo-PPA, it reduces the hardware cost of FloPoCo-PPA on average by 79% at 8-bit, 58% at 10-bit, and 9% at 12-bit resolutions. We finally show the benefits of our method by implementing a 10-bit homomorphic filter, which is used in image processing applications. Our method can implement the filter with no quality loss at lower hardware cost than what the previous approximate and exact methods can achieve.**

I. Introduction

The binary representation has 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].

Pure unary (PU) [2], [3] was introduced as a way of implementing non-linear 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 [4], [5], [6], [7] in terms of the area $\times$ delay hardware cost [3]. 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) [8], [9] was proposed to take advantage of unary and binary to make the method more scalable. It breaks a function into sub-functions with limited input and output ranges and implements them efficiently based on the PU method. Therefore, the lower bits of the input compute the sub-functions 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 $\times$ delay hardware cost compared to PU.

FloPoCo[1] [10] is a generator of arithmetic cores for FPGAs. One method it uses to approximately implement arbitrary functions is piece-wise polynomial approximation (PPA) [11]. It divides the input interval of a function into several sub-intervals that are addressed by the higher bits of the input. Given a target maximum error, each sub-function in this method is approximated by a polynomial with the Horner scheme.

In this paper, we propose a novel method to optimize the hardware cost of HBU using self-similarities among the sub-functions. Given a target maximum error, it makes pair-wise comparisons to figure out which sub-functions can be derived from each other through simple bit-wise transformations. For instance, if a sub-function g_i is similar to the inverse of a sub-function g_j, then we can ignore the implementation of g_i in unary and use NOT gates to derive it from g_j. This practice replaces the unary sub-functions (including their scaling networks and decoders) with simple post-processing logic gates to reduce the hardware cost. After finding the derivable sub-functions, our method tries to find the minimum set of them that can derive all the other sub-functions.

By implementing several non-linear functions at 8-, 10-, and 12-bit resolutions, our results show that our method outperforms HBU and FloPoCo-PPA in terms of area $\times$ delay hardware cost. Given the approximation of least significant bit,

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

it improves the hardware cost of HBU on average by 21% at 8-bit, 49% at 10-bit, and 60% at 12-bit resolutions. Compared to FloPoCo-PPA using the same approximation error budget, our method improves the hardware cost by 79% at 8-bit, 58% at 10-bit, and 9% at 12-bit resolutions. Additionally, if no approximation is allowed, which is a tough restriction on our proposed algorithm, our method still outperforms HBU on average by 7% at 8-bit, 20% at 10-bit, and 35% at 12-bit resolutions. By implementing a 10-bit homomorphic filter, we also show that our method can implement the filter with no quality loss at lower hardware cost compared to HBU and FloPoCo-PPA.

Our recent work [12] proposed a very rudimentary version of the similarity metric used in this work. In our rudimentary work, we had a strict metric for similarity, which left a lot of optimization on the table. We also explored pair-wise similarity checks locally, as opposed to a more global method used in this work. Furthermore, our previous work focused 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.

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

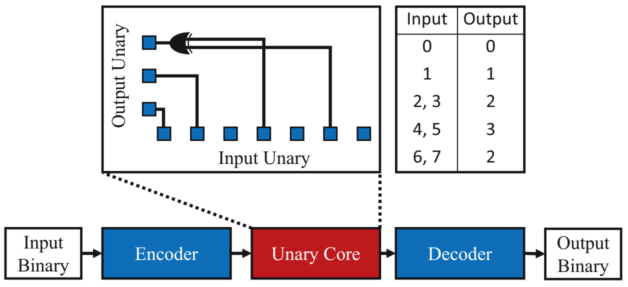
- Our method changes the way HBU breaks up functions. HBU breaks up functions into non-uniform input-range sub-functions, with the sole goal of reducing hardware cost. Our method forces an equal input range for all sub-functions, with the hope of deriving many of them from a core set of sub-functions.
- Our method uses a number of linear transformations to check if a sub-function can be derived from another one. This is in contrast to [12] that only looked at non-transformed matches within the approximation tolerance.
- Instead of pair-wise, local comparison between sub-functions [12], our method uses a better optimization method to find the best subset of sub-functions 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 work could reduce the total number of sub-functions from 512 to 218, whereas our new method can reduce them from 512 to 71 unique sub-functions.
- Our method provides HBU with a trade-off between accuracy in terms of maximum error and hardware cost in terms of area $\times$ delay.
- It outperforms the hardware cost of HBU even without any loss of accuracy.
- 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 parts of our method. Section II-B introduces our proposed method and algorithm, followed by a guiding example in Section II-C. In Section III, the HBU, FloPoCo-PPA, and our proposed method are compared on several functions at 8-, 10-, and 12-bit resolutions. The benefits of our method are evaluated in the implementation of a 10-bit homomorphic filter as an image processing application in Section IV. Finally, Section V concludes the paper and results.

II. Methodology

A. Previous Unary Works

PU (pure unary) [3] 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 binary, the unary core maps the input unary to output unary through the wires and XOR gates to implement the desired 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 a look-up table. Note that the function is increasing from $x = 0$ to $x = 5$, and then decreasing from $x = 6$ to $x = 7$. The XOR gate handles the non-monotonic section of the function. In unary methods, the input and output values of a function are considered unsigned integers, although one can change the interpretation of the raw values to represent signed integers or signed / unsigned fixed-point values.



Input	Output
0	0
1	1
2, 3	2
4, 5	3
6, 7	2

Fig. 1: Hardware architecture of PU [3] for an arbitrary function.

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 [9], on the other hand, uses the lower bits of the input to perform some parts of computation in unary and uses the higher bits of the input to complete the computation in binary. It breaks a function $f(x)$ into several sub-functions $g_i(x)$. The 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 smaller unary-to-binary encoders would be needed to convert the output back to binary. Each sub-function would be added to a bias value b_i to reconstruct the original function $f(x)$. The addition operation is performed in binary.

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

As mentioned above, in contrast to $f(x)$, a sub-function $g_i(x)$ has a limited input and output range; therefore, it can be efficiently implemented using the PU method [3], leading to a more scalable method. All these sub-functions 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 sub-function is going to be used and multiplexes the corresponding bias from a binary look-up table. Finally, a binary adder adds the sub-function and the bias together to compute the final output. Fig. 2 shows the overall architecture of HBU.

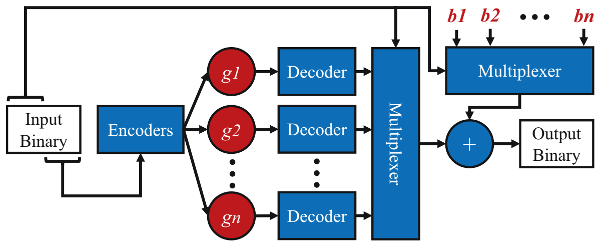


Fig. 2: Hardware architecture of HBU [9].

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 and considers sub-functions' slopes and output ranges to break a function hierarchically. The sub-functions 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 $n = 2^{w_b}$ sub-functions and separate their initial biases. For a w-bit function, the input range of all the sub-functions is $w_u = w - w_b$ bits.

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

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 pair-wise similarities between sub-functions to find the minimum set of unique sub-functions, from which all the other sub-functions can be derived using a set of bit-wise transformations. 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 bit-wise 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 the rounding to the nearest integer function. Intuitively, bm_i tries to "center" the transformed sub-function around the center of gravity of the target sub-function to reduce the amount of error in deriving the sub-function. This constant cannot get any arbitrary value and there are constraints on that, as described in Eq. 3.

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

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

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

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

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

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

$$\exists T_i, Err(g_i, T_i, g_j) \le TargetErr \\ \Rightarrow (b_i + g_i(x)) \simeq (b_i + bm_i) + T_i\{g_j(x)\} \\ \Rightarrow f_i(x) \simeq \hat{b}_i + T_i\{g_j(x)\}$$

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

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

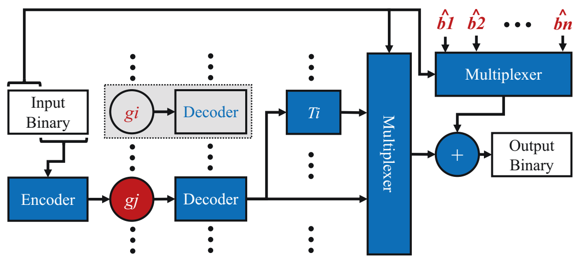


Fig. 3: Hardware architecture of our proposed method.

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 look-up table and transforming the sub-function $g_j(x)$, 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 sub-functions, 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 sub-functions 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 sub-function. We continue this process until all the entries of the similarity matrix get zero. We do understand that dynamic programming could have been used to get a globally minimum number of unique sub-functions, but in this version, we have limited ourselves to this greedy approach.

Algorithm 1 describes the procedure in more detail. To represent the final set of unique sub-functions 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 sub-function's output values and bias.

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}

Algorithm 1: SimBU Algorithm

```
Parameters: $TargetErr, w, w_b, w_u$
Input: $F = \{y = f(x) | x, y \in \mathbb{Z}$ and $x, y \in [0, 2^w)\}$
Outputs: $Sub, Bias, Unique, Transformer$
# Uniformly breaking the function into sub-functions
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 pair-wise comparisons of the sub-functions
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 sub-functions
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
```

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

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

Next, we make pair-wise comparisons to find all the sub-functions that can be derived from each other. We can use the

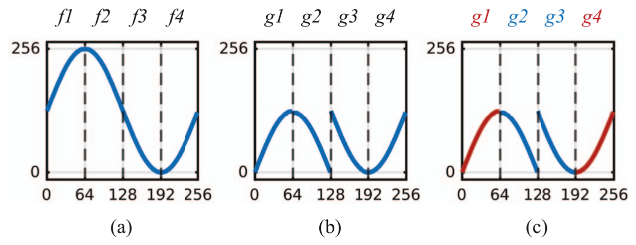


Fig. 4: Sub-functions of the guiding example. The red sub-functions are unique sub-functions that derive the others.

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 $g_2(x)$.

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

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 $Err(g_4, T_4, g_2) \le TargetError = 2^{-7}$. For this reason, similarity matrices are not necessarily symmetric.

After finding the derivable sub-functions, 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 sub-function. 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 sub-function. Then, by traversing through the 1st column of SM, we can see $SM[3][1] = 1$. As a result:

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

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 sub-function to be implemented. Similarly, we traverse through the 1st column and see $SM[2][4] = 1$. As a result:

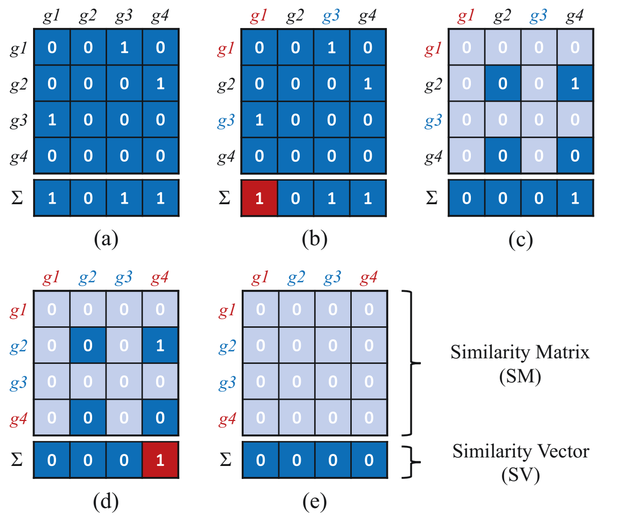
$$\begin{aligned} SM[2][4] = 1 &\Rightarrow Err(g_2, inv, g_1) = 0.0058 \le 2^{-7} \\ &\Rightarrow (131 + g_2(x)) \simeq (131 - 3) + inv\{g_4(x)\} \\ &\Rightarrow f_2(x) \simeq 128 + inv\{g_4(x)\} \end{aligned}$$

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 sub-functions (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 sub-functions and $g_2(x)$ and $f_3(x)$ are derived from them.



g1 g2 g3 g4
Σ
(a)
(b)
(c)
(d)
(e)
Similarity Matrix (SM)
Similarity Vector (SV)

(a)

	g1	g2	g3	g4
g1	0	0	1	0
g2	0	0	0	1
g3	1	0	0	0
g4	0	0	0	0
Σ	1	0	1	1

(b)

	g1	g2	g3	g4
g1	0	0	1	0
g2	0	0	0	1
g3	1	0	0	0
g4	0	0	0	0
Σ	1	0	1	1

(c)

	g1	g2	g3	g4
g1	0	0	0	0
g2	0	0	0	1
g3	0	0	0	0
g4	0	0	0	0
Σ	0	0	0	1

(d)

	g1	g2	g3	g4
g1	0	0	0	0
g2	0	0	0	1
g3	0	0	0	0
g4	0	0	0	0
Σ	0	0	0	1

(e)

	g1	g2	g3	g4
g1	0	0	0	0
g2	0	0	0	0
g3	0	0	0	0
g4	0	0	0	0
Σ	0	0	0	0

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

III. Implementation Results

We evaluated our method and compared it to the previous works including HBU [9] and FloPoCo-PPA [10], [11]. 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. 6 shows the graph of the implemented functions.

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 piece-wise polynomial approximation with the Horner scheme.

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

TABLE I: Equations of the implemented non-linear 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

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 $\Leftrightarrow TargetError = 2^{-w}$

Table II shows the hardware cost and accuracy of each implemented function using 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 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 sub-functions after the basic bit-wise transformations. That is, the sub-function 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 sub-functions at higher resolutions, and our method can reduce the number of unique sub-functions significantly, which compensates the added hardware resource

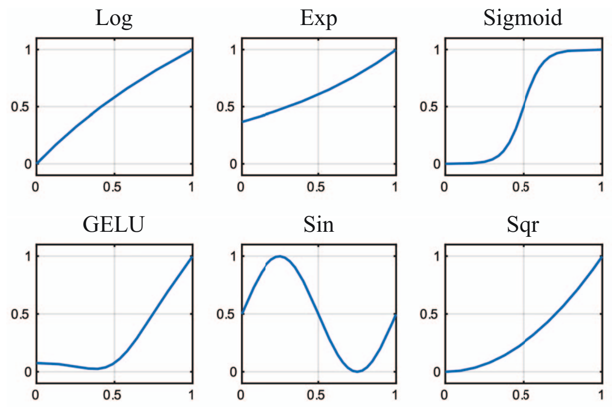


Fig. 6: Graphs of the implemented functions.

for implementing the transformations T_i. It can be seen from the table that SimBU-Exact reduces the area × delay cost of HBU on average by 7% at 8-bit, 20% at 10-bit, and 35% at 12-bit resolutions. Table III also shows the number of sub-functions before and after the self-similarity measures using our proposed method. The total number of sub-functions 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 HBU on average by 21% at 8-bit, 49% at 10-bit, and 60% at 12-bit resolutions. It improves the hardware cost of FloPoCo-PPA on average by 79% at 8-bit, 58% at 10-bit, and 9% at 12-bit resolutions. As seen in the table, our method fully utilizes the error budget to simplify the hardware architecture and reduce 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 [9] show that HBU performs better than the conventional binary, PU [3], and stochastic computing methods [4], [5], [6], [7] 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.

IV. Application

In PET or CT scans, multiplicative noise can obscure the image, making it harder to distinguish relevant features. Homomorphic filtering is a common technique that can filter out the multiplicative noise while also fixing the dynamic

TABLE II: Non-linear functions' hardware cost and accuracy results.

Exact Methods

HBU [9]	**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 HBU			**7.48%**					**19.72%**					**35.11%**		

Approximate Methods

FloPoCo-PPA [10], [11]	**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 HBU			**20.97%**					**49.15%**					**60.30%**		
Improvement Over FloPoCo-PPA			**79.20%**					**58.31%**					**8.51%**		

range issue and increasing contrast. An example of this is shown in Fig. 8a, where the full-body PET scan shows two hot spots. These dominate the dynamic range, hence reducing detail on other body features. After homomorphic filtering, we get Fig. 8b. This retains the two hot spots in the brain and the lung but increases the detail on lower-intensity parts of the image. Another application of homomorphic filtering is in neurocomputing to decode information from the spiking sequence of a neuron model [13].

For the purposes of this work, we will investigate homomorphic filtering as an image enhancement technique. Here an input image follows the illumination-reflectance model [14], where the image is decomposed as a product of the illumination $i(x, y)$ and reflectance $r(x, y)$.

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

By applying the log transformation, we can represent the image as the sum, instead of the product, of illumination and reflectance, hence allowing us to filter noise from these two

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

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

components.

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

The filtering part of this process is typically done in the frequency domain by applying FFT to the log-transformed image. But for this FPGA application, the filtering is done in the spatial domain by convolving a 5×5 high-pass filter kernel. To test our implementation of non-linear functions, we implemented an end-to-end flow for a 5×5 window, which will loop across the entire image. A block diagram of this is shown in Fig. 7. The Log and Exp layers perform the following functions:

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

To support FloPoCo-PPA [10], [11] and get good accuracy, the functions and convolution output are re-scaled and shifted such that the input and output ranges of each non-linear function fit the interval $[0, 1)$. Since each input and output scale is a predetermined constant, we do not require additional multiplication/division to scale up/down our quantized values, and these scaling constants are built into the non-linear functions. Since the input and output of each non-linear function are in the range of $[0, 1)$, we don't require any integer bits in the fixed point representation of the inputs and outputs in the functions. Since convolution outputs a signed fixed-point integer, we add and shift to convert the range to $[0, 1)$ before passing through the Exp layer, in which $2x - 1$ corrects for the range change. The bit lengths of multiplication and accumulation in convolution are automatically extended to ensure no overflow. The re-scale and shift after the convolution is needed to bring the bit length back down to the same range as the Exp layer. Re-scaling is done purely through a right-shift operation.

To evaluate our method in terms of hardware cost and accuracy, we used Xilinx's Kintex-7 FPGA and Vivado 2020.2

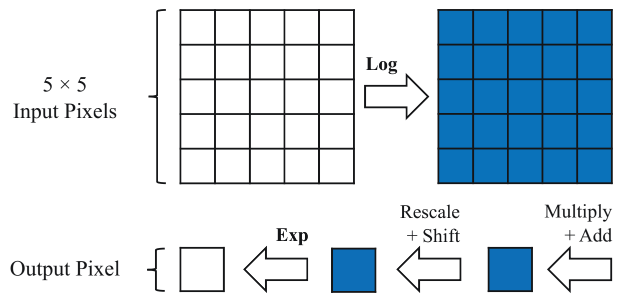


Fig. 7: Block diagram of the homomorphic filter on a 5×5 window.

to implement a 10-bit homomorphic filter for a 5×5 window, as described in Fig 7. For accuracy results, the final image using the quantized look-up tables of the Log and Exp layers, and integer-based convolution is compared to the floating-point reference output. Table IV shows the hardware cost and accuracy of the implemented homomorphic filtering. Since the function in the Exp layer is only done once to the output pixel, we implemented this layer using SimHBU-Exact for both variants of SimHBU. As seen, our SimBU-Exact method improves the area $\times$ delay hardware cost of HBU by 7% without losing quality, and our SimBU-Approx method improves that by 44% with negligible quality loss. Compared to FloPoCo-PPA, our SimBU-Approx method reduces the hardware cost by 46% with higher quality. We can see in Fig. 8c that the subjective quality difference between SimBU-Approx and reference image is well within the acceptable range. Finally, Fig. 9 compares the area $\times$ delay hardware cost of the implemented non-linear functions and homomorphic filtering.

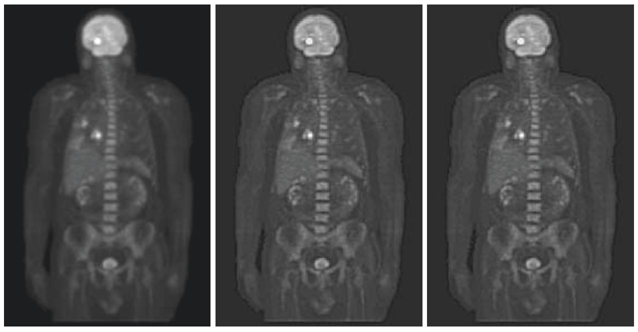


Fig. 8: Homomorphic filtering test images of PET Scan. Test image sourced from [14]. Part (b) shows homomorphic filtering applied to the input image using floating point computations. Part (c) shows the results from homomorphic filtering using the SimBU-Approx method.

V. Conclusion

In this work, we proposed a method to implement non-linear functions given a target maximum error. It provides a

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

Exact Methods					
Method	**Area**	**Delay**	**A × D**	**MSE**	**PSNR**
HBU [9]	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 [10], [11]	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

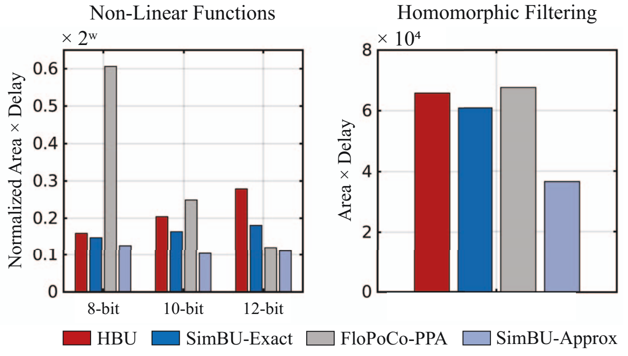


Fig. 9: Comparisons of hardware cost in non-linear functions and homomorphic filtering.

trade-off between accuracy and hardware cost by reducing the number of sub-functions in the previous HBU (hybrid binary-unary) method and replacing them with simple bit-wise transformers. In terms of area × delay hardware cost, our results show that our method outperforms HBU even with no approximation error budget. By approximating the least significant bit, our method beats the FloPoCo-PPA (piece-wise polynomial approximation) method at up to 12-bit resolutions as well. Finally, we implemented a 10-bit homomorphic filter as an image processing application to show the benefits of our method compared to the previous works. Without loss of quality, our method implemented the filter at lower hardware cost compared to the previous exact and approximate methods.

Acknowledgment

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] 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

[3] 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

[4] 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.

[5] 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

[6] 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

[7] 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.

[8] 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

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

[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] 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.

[12] 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

[13] 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.

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