Constant Coefficient Multipliers Using Self-Similarity-Based Hybrid Binary-Unary Computing

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

2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323844

Abstract—**Constant coefficient multipliers are widely used in digital signal processing and machine learning architectures. Researchers have proposed HBU-CCM (hybrid binary-unary constant coefficient multiplier), which is an approximate method that outperforms conventional binary and FloPoCo-KCM (table-based real multiplier) methods in terms of hardware cost at the expense of accuracy due to aliasing issues. SimBU (self-similarity-based hybrid binary-unary) is another method that was recently proposed to implement general nonlinear functions using self-similarities leading to few hardware resources. In this work, we use a simplified version of the SimBU algorithm to address the aliasing issues of HBU-CCM and improve accuracy. We also implement a convolution kernel for a Gaussian blurring filter to evaluate our method and compare it to previous works. Our method outperforms conventional binary and FloPoCo-KCM methods in terms of hardware cost with desired accuracy and with no aliasing error as opposed to HBU-CCM.**

1. Introduction

Multiplication is ubiquitous in digital systems and is used in many applications such as signal processing, machine learning, and machine vision. For instance, multiplication takes 90% of the computation time in a convolutional neural network (CNN) in which hundreds of thousands of parallel multiply-accumulate operations are performed [1], [2], [3]. However, due to limited hardware resources, implementing that many multipliers in parallel is not feasible in most FPGAs [4]. Additionally, folding the architecture of a CNN requires on-chip or off-chip memory to store weights, which limits the efficiency significantly [4]. Using high-performance constant coefficient multipliers (CCMs) is a promising solution that can directly perform the multiplications without transferring the weights between memory and multipliers [5]. Recently, various methods have been proposed to optimize CCMs in terms of hardware cost and accuracy [6], [7], [8], [9], [10].

FloPoCo [7] is a framework to generate arithmetic cores on FPGAs that encompasses a variety of methods to perform linear and nonlinear operations. It can implement CCMs with real coefficients using a table-based method, which is referred to as FloPoCo-KCM. In this method, pre-computed partial products are stored in lookup tables and added together base on the input value [8]. Despite the simplicity of this method, it utilizes a large amount of hardware resources for storing the partial products, especially at high resolutions and low target error values.

Unary computing uses thermometer encoding of numbers, which in turn paves the way for performing computations needed to evaluate an arbitrary function using only wires and XOR gates instead of complex logic gates [11], [12], [13], [14], [15], [16]. Pure uanry (PU) [11], [12] is the first method that used unary representations to implement such nonlinear functions at less hardware cost than conventional binary and stochastic computing methods [17], [18], [19], [20]. However, PU is not efficient at high resolutions, because it is not scalable, and its hardware complexity grows exponentially as the resolution increases. Hybrid binary-unary (HBU) [13], [14] method was then proposed to solve the scalability issue of PU by breaking a function into sub-functions at lower resolutions and implementing them using the PU method at lower hardware cost. Although HBU is more scalable compared to PU, it needs to implement and multiplex a large number of sub-functions. However, if the function of a multiplier with real constant coefficient is broken into sub-functions, they will look very similar to each other. They are not exact matches of each other as a byproduct of quantization, though. Hybrid binary-unary constant coefficient multiplier (HBU-CCM) [9], [10] applies such approximations to use only one sub-function to implement a CCM, which results in reductions in hardware cost at the expense of accuracy.

Self-similarity-based hybrid binary-unary computing (SimBU) [16] was recently proposed to use self-similarities in a nonlinear function to reduce the number of sub-functions needed to implement the function at the desired precision level. In other words, it first identifies the similar sub-functions that can be derived from each other through simple transformations such as right/left shifting, inverting, and vertical moving. Then, it chooses a minimum set of unique sub-functions that can derive the original function given the target accuracy. In this paper, we use a simplified version of the SimBU algorithm to implement CCMs at higher accuracy than HBU-CCM. It not only limits the maximum absolute error according to the given target error value, but also minimizes the mean square error. We implemented a number of CCMs with real coefficients generated by SimBU at different resolutions and compared them to FloPoCo-KCM using the same target error values. Also, we deployed the SimBU-CCM method to implement a convolution kernel for a Gaussian blurring filter as an image processing application to investigate the effects of aliasing error caused by HBU-CCM and compare our method to previous works such as conventional binary, FloPoCo-KCM, and HBU-CCM in terms of hardware cost and accuracy.

The rest of the paper is as follows. In Section 2.1, we discuss the previous unary works and their pros and cons. Next, we introduce a simplified version of the SimBU algorithm for constant coefficient multipliers in Section 2.2. We present the FPGA implementation results of constant coefficient multipliers in Section 3, followed by implementation results of a convolution kernel for a Gaussian blurring filter in Section 4. Finally, we conclude the paper in Section 5.

2. Methodology

2.1. Previous Unary Works

PU (pure unary) [11] was proposed as an alternative to conventional binary and stochastic computing methods to implement math functions. PU implements the core of functions in hardware using the unary number representation. In unary, a decimal value m is represented as a sequence in which the first m bits are 1's and the rest are 0's. The total length of the sequence depends on the maximum value that is needed to be represented in a system. For instance, the decimal value 4 is 100 in binary and is 1111000 in unary. Performing computations on unary numbers on FPGAs is much simpler than conventional binary due to the unpacked nature of the number encoding. For a constant
coefficient multiplier, the computation can be perf
using a network of wires, which is called the ”unary
Fig. 1 shows the architecture of PU and the unary c
$f(x) = \frac{1}{3}x$ as an example. The figure shows how an
value of 3 is converted to an output value of 1. To
PU computations consistent with other computation
digital system, which are performed in binary, an e
and decoder are used in the PU architecture to conv
numbers from binary to unary and back.

Other multipliers with real coefficients can be i
mented using wires in unary similar to the unary
in Fig. 1. For any real coefficient (e.g., 0.753 or *lo*
the truth table of the multiplication can be obtained
quantizing the input and output values into fixed-poi
resentations. Then, the resulting truth table can be en
into unary codes and implemented using the PU met

Although the unary cores are simple, the encode
decoders are costly, especially at high resolutions. I
the cost of these units grows exponentially due to th
that the length of a unary sequence grows expone

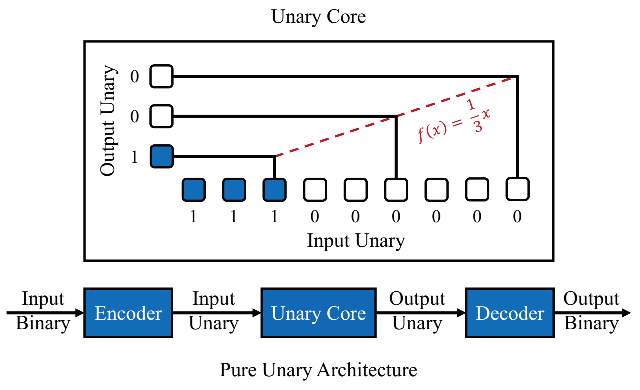


Figure 1: Hardware architecture of a PU (pure unary) unit.

as the resolution increases, which in turn makes PU not scalable. To tackle the scalability issue of PU, HBU (hybrid binary unary) [13] was proposed. HBU breaks a function into a number of sub-functions. It also subtracts the minimum value of the sub-function, which is called the bias of that sub-function, from all output values in that sub-function to limit its output range. As a result, a function f turns into n sub-functions g_i with shorter input and output ranges compared to the original function f.

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

By shortening the input and output ranges, each sub-function g_i can be implemented using the PU method with a less complex encoder and decoder. In HBU, all the sub-functions are implemented using PU and performed in parallel. The lower bits of the input binary are enough to perform the computations of the sub-functions. However, since the output of only one sub-function is desired based on the input, the higher bits of the input binary are needed

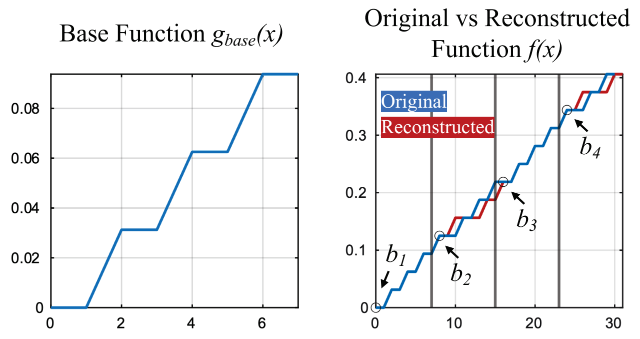


Figure 2: The aliasing error introduced after the reconstruction of a CCM using the HBU-CCM method.

In CCMs with real coefficients, the sub-functions are approximately equal to each other, i.e., $g_1 \approx g_2 \approx \cdots g_n$.

Such approximations are referred to as ”aliasing”.
the sub-functions are exactly the same when re
as real numbers, this is not the case after a fi
quantization. HBU-CCM [9] is a method that use
approximations to implement such multipliers less
other words, a function $f(x) = cx$ is written as f

$$f(x) \approx g_{base}(x) + \begin{cases} b_1 & x \in [0, x_1) \\ \dots \end{cases}$$

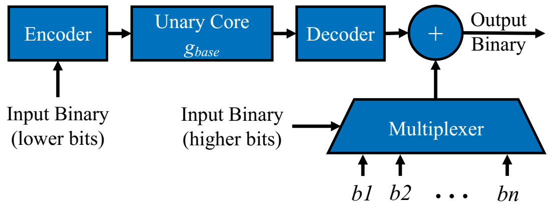


Figure 3: The hardware architecture of HBU-CCM (hybrid binary-unary constant coefficient multiplier).

The aliasing error can result in one bit inaccuracy and potentially affect the quality of a system dramatically. In Section 4, we show how such approximations affect the accuracy of a convolution operation in an image processing application. To address the aliasing issue, a simplified version of the SimBU [16] algorithm can be deployed. The original SimBU algorithm was proposed to implement nonlinear functions by breaking a function into sub-functions and exploiting the self-similarities among them and their transformations. For instance, function $f(x) = sin(2\pi x)$ is uniformly broken into four sub-functions, however, in SimBU as opposed to HBU, only two of them are implemented using the PU method. The other sub-functions are derived from those two sub-functions through NOT gates. In this paper, we use the SimBU algorithm to find a minimum set of sub-functions to implement a CCM with desired accuracy and with no aliasing error. The original SimBU algorithm provides a trade-off between hardware cost and accuracy, which would be beneficial for many applications, but its applicability to CCMs is limited due to the aliasing issue. We harness the hardware saving benefits of SimBU, but further customize the algorithm to specifically address the aliasing issue in CCMs, which are widely used in machine learning and image processing applications.

2.2. Simplified SimBU [16] Algorithm for CCMs

As we discussed in Section 2.1, HBU-CCM [9] breaks the function of a CCM into sub-functions and approximates

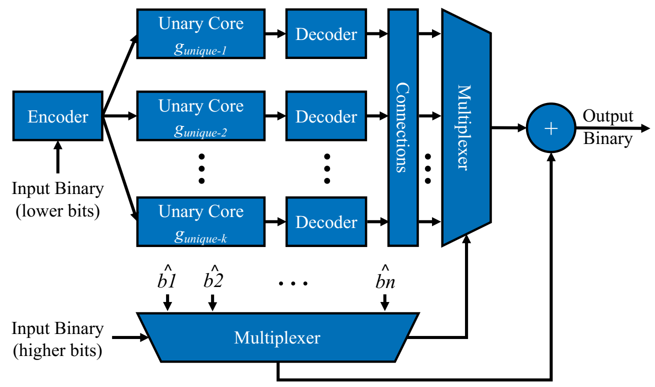

Encoder
Input Binary (lower bits)
Unary Core $g_{unique\text{-}1}$
Unary Core $g_{unique\text{-}2}$
Unary Core $g_{unique\text{-}k}$
Decoder
Decoder
Decoder
Connections
Multiplexer
+
Output Binary
$\hat{b1}$ $\hat{b2}$ ... $\hat{bn}$
Input Binary (higher bits)
Multiplexer

Figure 4: The hardware architecture of SimBU-CCM (our method). The modified biases can be obtained by Eq. 6

all of them with a single sub-function g_{base}, which creates aliasing errors. In this section, we introduce a simplified version of the SimBU [16] algorithm to implement CCMs given a target maximum absolute error.

We uniformly break a function $f(x) = cx$, where c is a real constant value, into n sub-functions.

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

where w_u is the bit-length of the sub-functions g_i. Therefore, the total number of sub-functions is $n = 2^{w-w_u}$, where w is the bit-length of the original function f. In Fig. 2, $n = 4$.

Next, we separate the initial bias of each sub-function to reduce their output ranges. The initial bias is the minimum value of each sub-function in its input interval.

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

where $b_i = min\{f_i\}$ is the initial bias of the sub-function f_i. The bias values $b_1, b_2, \cdots, b_4$ are shown in Fig. 2.

Unlike the previous work HBU-CCM [9], we do not approximate all the sub-functions into a single sub-function $g_{base} \approx g_1 \approx \cdots \approx g_n$. Doing so will introduce aliasing errors, as evident in Fig. 2: using only g_{base} to approximate all sub-functions results in mismatches between the quantized value of the function and the approximate version. Instead, we compare all the sub-functions together and find the minimum set of unique sub-functions that can reconstruct the original function f given the target error value. In the example of Fig. 2, f_2 can better approximate f_4 than f_1: it would results in less aliasing mismatches.

We also modify the initial biases b_i to maximize the similarities among the sub-functions and minimize the hardware cost, which is something the original SimBU work [16] did not do. Furthermore, unlike the original SimBU algorithm, we do not need to consider transformations such as NOT, shift-left and shift-right of each sub-function when searching for similarities among different sub-functions. That would have been beneficial in nonlinear functions.

To find the minimum set of sub-functions, we first identify the similar sub-functions and construct the following $n \times n$ Boolean matrix, which is called $SimilarityMatrix$ or SM.

$$SM = \begin{bmatrix} sm_{11} & sm_{12} & \cdots & sm_{1n} \\ sm_{21} & sm_{22} & \cdots & sm_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ sm_{n1} & sm_{n2} & \cdots & sm_{nn} \end{bmatrix}_{n\times n} \quad (1)$$

where sm_{ij} $(i \neq j)$ is 1, if and only if g_i can be derived from g_j given the target maximum absolute error $TargetErr$.

$$\begin{aligned} sm_{ij} = 1 \Leftrightarrow \exists\ bm_i, i \neq j, max\{|Float(b_i + g_{\mathbf{i}}(x)) \\ -Fixed(b_i + bm_i + g_{\mathbf{j}}(x))|\} \leq TargetErr \end{aligned} \quad (2)$$

where $Float(x)$ and $Fix(x)$ denote the floating-point and fixed-point value of x, respectively. The parameter bm_i is a constant value that modifies the initial bias b_i to increase the similarity between g_i and g_j. This parameter can be obtained by the following equation, which calculates the rounded average point-to-point distance between the two functions.

$$bm_i = \lfloor \frac{1}{2^{w_u}} \sum_x (g_i(x) - g_j(x)) \rceil \quad (3)$$

However, the parameter bm_i cannot take any arbitrary value, and there are constraints on that to make sure that the addition of the sub-function and the modified bias fits in w bits.

$$0 \leq bm_i + b_i + g_j(x) < 2^w \quad (4)$$

After finding all the similar sub-functions and constructing the matrix SM, we can find the minimum set of unique sub-functions through an iterative process. We add all the entries of SM vertically to obtain an n-element vector, which is called $SimilarityVector$ or SV.

$$SV = \begin{bmatrix} sv_1 & sv_2 & \cdots & sv_v \end{bmatrix}_{1\times n} \quad (5)$$

where $sv_j = \sum_{i=1}^{n} sm_{ij}$. The index of the maximum element of SV indicates the first unique sub-function, i.e. g_p is the first unique sub-function, where $p = \arg\max_{idx} SV[idx]$. All the non-zero entries in the p^{th} column indicate all the other sub-functions that can be derived from g_p. Next, we zero the p^{th} row and column of SM as well as all the rows and columns corresponding to the sub-functions that can be derived from g_p. For instance, if g_s can be derived from g_p (i.e., $sv_{sp} = 1$), we zero the s^{th} row and column of SM as well, because that sub-function g_s is no longer available as an independently-implemented sub-function for other sub-functions to be derived from it. Additionally, the initial bias b_s must be modified into $\hat{b}_s = b_s + bm_s$, where bm_s can be obtained by Eq. 3.

$$\hat{b}_s = b_s + bm_s \quad (6)$$

Next, we recalculate the vector SV and find the next unique sub-function. We continue this process until all the elements of SV are zeros.

As a result, a number of unique sub-functions are found that implement the function $f(x) = cx$ with the desired accuracy, as opposed to HBU-CCM which uses only one sub-function to approximately implement the function.

Algorithm 1 shows the simplified version of SimBU [16] for CCMs. Sub and $Bias$ are two arrays that store the values of the sub-functions and their initial biases, respectively. $UniqueSub$ is an array that shows the index of a unique sub-function for each sub-function. For instance, if $Unique[i] = j$, it means that sub-function g_i can be derived from the unique sub-function g_j. $ModifiedBias$ is another array that holds the initial bias of each sub-function after the modification due to the parameter bm_i according to Eq. 3.

Fig. 4 shows the architecture of SimBU for CCMs. The block "Connections" shares the outputs of the unique sub-functions to derive all the sub-functions according to their similarities that are found by the simplified SimBU algorithm.

Algorithm 1: Simplified SimBU [16] Algorithm

```
1  Parameters: $TargetErr$
2  Input: $Sub, Bias$
3  Outputs: $UniqueSub, ModifiedBias$
4  $n \leftarrow len(Subfunction)$
5  for $i = 1\ to\ n$ do
6      for $j = 1\ to\ n$ do
7          $bm_i \leftarrow \lfloor \frac{1}{2^{w_u}} \sum(Sub[i] - Sub[j]) \rceil$
8          $Err \leftarrow max\{|Float(Bias[i] + Sub[i]) -$
9                  $Fixed(Bias[i] + bm_i + Sub[j])|\}$
10         if $Err \leq TargetErr$ then
11             $SM[i][j] \leftarrow 1$
12         else
13             $SM[i][j] \leftarrow 0$
14         end
15     end
16 end
17 while $\sum_i \sum_j SM[i][j]\ != 0$ do
18     $SV \leftarrow \sum_i SM[i][:]$
19     $idx \leftarrow \arg\max_i SV[i]$
20     for $i = 1\ to\ n$ do
21         if $i\ != idx\ and\ SM[i][idx] == 1$ then
22             $Unique[i] \leftarrow idx$
23             $bm_i \leftarrow \lfloor \frac{1}{2^{w_u}} \sum(Sub[i] - Sub[idx]) \rceil$
24             $ModifiedBias[i] \leftarrow Bias[i] + bm_i$
25             $SM[i][:] \leftarrow 0$
26             $SM[:][i] \leftarrow 0$
27         end
28     end
29     $UniqueSub[idx] \leftarrow idx$
30     $SM[idx][:] \leftarrow 0$
31     $SM[:][idx] \leftarrow 0$
32 end
```

TABLE 1: Average FPGA hardware cost of real CCMs, with randomly chosen constants in the range $[0, 1)$.

Method	TargetErr (ulp)	Bit-Width $w_{in} = 8, w_{out} = 8$				Bit-Width $w_{in} = 8, w_{out} = 16$			
		Area (LUT)	Delay (ns)	A × D	Ratio	Area (LUT)	Delay (ns)	A × D	Ratio
FloPoCo-KCM [7]	0.51	22.94	1.92	**44.34**	1	38.25	2.10	**81.09**	1
	0.75	13.53	1.76	**24.42**	1	29.34	2.00	**59.52**	1
	1.00	10.56	1.78	**20.40**	1	25.19	1.94	**49.01**	1
SimBU-CCM (our method)	0.51	14.06	1.84	**26.16**	0.59	24.22	2.07	**50.34**	0.62
	0.75	10.19	1.63	**17.88**	0.73	21.88	1.93	**42.52**	0.71
	1.00	8.00	1.48	**12.75**	0.62	20.72	1.95	**40.85**	0.83

3. Implementation Results

We developed a Matlab script to generate the RTL files of CCMs using our proposed method. The RealKCM tool in the FloPoCo[1] framework was also used to generate CCMs using the FloPoCo-KCM [7] method. For evaluation purposes, we generated 32 different multipliers with real coefficients, all of which were randomly chosen in the range $[0, 1)$. We do not expect the results to change significantly if one were to choose a different range, *e.g.*, $[0, 4)$ or $[-2, 3)$. The inputs of the multipliers were assumed to be 8-bit ($w_{in} = 8$), and the outputs were rounded to both 8-bit ($w_{out} = 8$) and 16-bit ($w_{out} = 16$) resolutions. We also considered 3 different target error values ($TargetError \in \{0.51ulp, 0.75ulp, 1.00ulp\}$) to design the multipliers and evaluate the trade-off between accuracy and hardware cost. The acronym *ulp* stands for "units in the last place". The multipliers with $TargetError = 0.5ulp$ introduce no approximation error other than the output quantization error which is inevitable in hardware. On the other hand, the multipliers with $TargetError = 1.00ulp$ introduce some error equal to the approximation of the least significant bit. In total, we generated 192 different multipliers using each method for the purpose of evaluation. The RTL files were synthesized on Xilinx's Kintex-7 FPGA using Vivado 2020.2.

Table 1 shows average implementation results of the multipliers using different methods at different output bit lengths and target error values. In all the tables and figures, "A × D" denotes the area × delay hardware cost. Fig. 5 shows the area × delay hardware cost for each coefficient. As seen, at 8-bit output resolution, our method outperforms FloPoCo-KCM by 41%, 27%, and 38% at target error $0.5ulp$, $0.75ulp$, and $1.00ulp$, respectively. At 16-bit output resolution, it outperforms FloPoCo-KCM by 38%, 29%, and 17% at target error $0.5ulp$, $0.75ulp$, and $1.00ulp$, respectively.

In SimBU, it is feasible to implement a multiplier using different target error values for different input regions. Some applications need to perform multiplications with higher precision in some input regions and with lower precision in some other regions. In such cases, $TargetError$ can be set

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

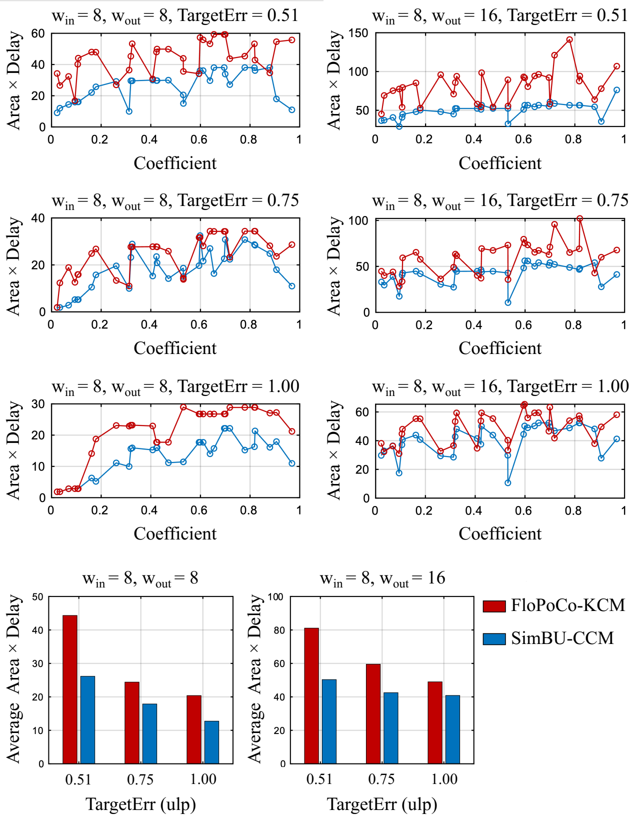


Figure 5: FPGA hardware cost of the real CCMs. Red graphs and bars represent FloPoCo-KCM, and blue bars and graphs represent our SimBU-CCM method.

to different values for different regions in the self-similarity measurements.

In this section, we did not compare our method to HBU-CCM, for HBU-CCM cannot implement CCMs given a target maximum absolute error. As the authors in [9] mention, the maximum absolute error of HBU-CCM without optimization is one bit (i.e., $1.00ulp$), and the maximum absolute error of HBU-CCM with optimization is even higher. As a result, this method cannot implement the multipliers with the same precision as our method and FloPoCo-KCM, and therefore, these methods are not comparable in terms of hardware cost. On the other hand, the hardware cost of our

TABLE 2: FPGA hardware cost and accuracy of the convolution kernel of the Gaussian blurring filter.

Method	TargetErr (ulp)	Area (LUT)	Delay (ns)	A × D	PSNR
Conventional Binary I	-	325	18.93	**6,152.25**	**45.01**
Conventional Binary II	-	231	17.21	**3,974.82**	**32.60**
FloPoCo-KCM [7]	0.51	507	5.63	**2,854.41**	**43.18**
	0.75	234	5.72	**1,338.95**	**38.60**
	1.00	165	4.80	**791.51**	**36.73**
HBU-CCM [9]	-	136	4.83	**656.61**	**34.32**
SimBU-CCM (our method)	0.51	292	5.36	**1,564.24**	**43.61**
	0.75	162	4.26	**689.47**	**42.16**
	1.00	160	4.26	**680.96**	**41.97**

method at large target error values is similar to that of HBU-CCM, because, at the large values, all the sub-functions are approximated to one sub-function as in the HBU-CCM method. Nonetheless, in Section 4, we will compare the accuracy and hardware cost of our method to HBU-CCM on the implementation of a convolution kernel for a Gaussian blurring filter.

4. Application

Convolution is a common operation performed in many applications such as machine learning and signal processing. In convolutional neural networks, for instance, this operation is performed in more than 90% of all operations [1], [2], [3]. In image processing, convolution is also used to filter an image in time domain.

Gaussian blurring is a low-pass filter that is mostly used in image processing. The formula of a Gaussian function in two dimension is defined as follows.

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} exp(-\frac{x^2+y^2}{2\sigma^2})$$

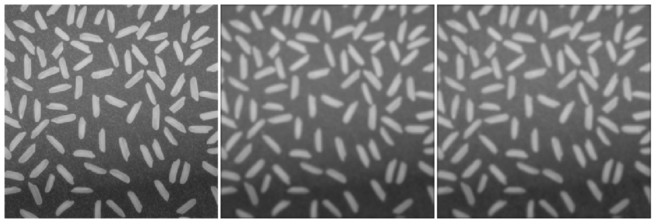
where x and y are the distances from the origin, and σ is the standard deviation of the Gaussian function. As a result, the convolution kernel for a 5×5 Gaussian blurring filter with $\sigma = 1.6$ is as follows.

$$G_{1.6} = \begin{bmatrix} 0.0165 & 0.0297 & 0.0361 & 0.0297 & 0.0165 \\ 0.0297 & 0.0534 & 0.0649 & 0.0534 & 0.0297 \\ 0.0361 & 0.0649 & 0.0789 & 0.0649 & 0.0361 \\ 0.0297 & 0.0534 & 0.0649 & 0.0534 & 0.0297 \\ 0.0165 & 0.0297 & 0.0361 & 0.0297 & 0.0165 \end{bmatrix}_{5\times5}$$

In this section, we deploy our method to implement the 5 × 5 convolution kernel for the Gaussian blurring filter. We also implemented the kernel using other methods such as conventional binary, FloPoCo-KCM [7], and HBU-CCM [9]. We synthesized the kernels on Xilinx's Kintex-7 FPGA using Vivado 2020.2.

As opposed to the conventional binary and HBU-CCM methods, our method and FloPoCo-KCM can implement the multipliers given a target error value. Therefore, we considered 3 different values as $TargetErr$ which resulted in different output image qualities. In conventional binary, we had to quantize the coefficient of the kernels in a fixed-point representation. Therefore, we considered 2 different bit lengths for the quantization process. Conventional Binary I denotes the design in which the coefficients were quantized into 8 fractional bits, whereas Conventional Binary II denotes the design in which the coefficients were quantized into 6 fractional bits. However, in all the designs, the final output after the multiply-add operations was shifted to fit in an 8-bit unsigned integer.

Table 2 shows the FPGA hardware cost and accuracy results of the convolution kernel for the Gaussian blurring filter. It can be seen that our method provides a clear advantage over previous works in terms of accuracy and hardware cost. For example, at similar PSNR values (43.18 and 43.61), our method provides a 45% reduction in area × delay hardware cost compared to FloPoCo-KCM. Compared to HBU-CCM, our method provides much better PSNR values.



(a) Input (b) Floating-Point (c) SimBU-CCM

Figure 6: The input (a) and output of the convolution kernel for the Gaussian blurring filter using floating-point operations (b) and our SimBU-CCM method with the target error $1.00ulp$ (c).

5. Conclusions

We used the SimBU algorithm to implement constant coefficient multipliers given a target maximum absolute error using few hardware resources. At the same target error values, our method outperforms the previous FloPoCo-KCM method in terms of hardware cost. On average, it reduces the hardware cost by 36% at 8-bit and 29% at 16-bit output resolutions compared to FloPoCo-KCM. We also implemented a convolution kernel for a Gaussian blurring filter to evaluate our method in terms of hardware cost and accuracy. Compared to conventional binary, our method reduces the hardware cost by 83% and increases the accuracy by 29%. Compared to FloPoCo-KCM, our method reduces the hardware cost by 49% and increases the accuracy by 9%. Compared to HBU-CCM, our method increases the hardware cost by 3% but increases the accuracy by 22%.

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] J. Garland and D. Gregg, "Low complexity multiply-accumulate units for convolutional neural networks with weight-sharing," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, sep 2018. [Online]. Available: https://doi.org/10.1145/3233300

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[3] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170. [Online]. Available: https://doi.org/10.1145/2684746.2689060

[4] S. Mittal, "A survey of fpga-based accelerators for convolutional neural networks," *Neural computing and applications*, vol. 32, no. 4, pp. 1109–1139, 2020.

[5] S. R. Faraji, P. Abillama, G. Singh, and K. Bazargan, "Hbucnna: Hybrid binary-unary convolutional neural network accelerator," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5.

[6] M. Kumm, O. Gustafsson, M. Garrido, and P. Zipf, "Optimal single constant multiplication using ternary adders," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 7, pp. 928–932, 2018.

[7] 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.

[8] E. G. Walters, "Reduced-area constant-coefficient and multiple-constant multipliers for xilinx fpgas with 6-input luts," *Electronics*, vol. 6, p. 101, 2017.

[9] S. R. Faraji, P. Abillama, and K. Bazargan, "Low-cost approximate constant coefficient hybrid binary-unary multiplier for dsp applications," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 93–101.

[10] ——, "Approximate constant-coefficient multiplication using hybrid binary-unary computing for fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 3, dec 2022. [Online]. Available: https://doi.org/10.1145/3494570

[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] 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

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

[15] 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

[16] ——, "Optimizing hybrid binary-unary hardware accelerators using self-similarity measures," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023, pp. 105–113.

[17] 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.

[18] 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.

[19] 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

[20] 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