CompressedLUT: An Open Source Tool for Lossless Compression of Lookup Tables for Function Evaluation and Beyond

Alireza Khataei
Department of Electrical and Computer Engineering
University of Minnesota
Minneapolis, MN, USA
khata014@umn.edu

Kia Bazargan
Department of Electrical and Computer Engineering
University of Minnesota
Minneapolis, MN, USA
kia@umn.edu

ABSTRACT

Lookup tables are widely used in hardware to store arrays of constant values. For instance, complex mathematical functions in hardware are typically implemented through table-based methods such as plain tabulation, piecewise linear approximation, and bipartite or multipartite table methods, which primarily rely on lookup tables to evaluate the functions. Storing extensive tables of constant values, however, can lead to excessive hardware costs in resource-constrained edge devices such as FPGAs. In this paper, we propose a method, called CompressedLUT, as a lossless compression scheme to compress arrays of arbitrary data, implemented as lookup tables. Our method exploits decomposition, self-similarities, higher-bit compression, and multilevel compression techniques to maximize table size savings with no accuracy loss. CompressedLUT uses addition and arithmetic right shift beside several small lookup tables to retrieve original data during the decoding phase. Using such cost-effective elements helps our method use low area and deliver high throughput. For evaluation purposes, we compressed a number of different lookup tables, either obtained by direct tabulation of 12-bit elementary functions or generated by other table-based methods for approximating functions at higher resolutions, such as multipartite table method at 24-bit, piecewise polynomial approximation method at 36-bit, and hls4ml library at 18-bit resolutions. We implemented the compressed tables on FPGAs using HLS to show the efficiency of our method in terms of hardware costs compared to previous works. Our method demonstrated 60% table size compression and achieved 2.33 times higher throughput per slice than conventional implementations on average. In comparison, previous TwoTable and LDTC works compressed the lookup tables on average by 33% and 37%, which resulted in 1.63 and 1.29 times higher throughput than the conventional implementations, respectively. CompressedLUT is available as an open source tool.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; **High-level and register-transfer level synthesis**.

KEYWORDS

Hardware Acceleration, High-Level Synthesis, Lookup Table, Lossless Compression, Table Size Reduction, Function Evaluation, Table-Based Method

ACM Reference Format:
Alireza Khataei and Kia Bazargan. 2024. CompressedLUT: An Open Source Tool for Lossless Compression of Lookup Tables for Function Evaluation and Beyond. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3626202.3637575

FPGA '24, March 3–5, 2024, Monterey, CA, USA

ACM ISBN 979-8-4007-0418-5/24/03…$15.00
https://doi.org/10.1145/3626202.3637575

1 INTRODUCTION

Lookup tables are widely used in both hardware and software systems to store blocks of read-only, predefined data. Such tables are used in programmable gate arrays (FPGAs), graphics processing units (GPUs), and digital signal processors (DSPs). Compressing lookup tables can potentially reduce their implementation costs in terms of memory resource utilization, throughput, power consumption, *etc.* This issue has been of considerable interest as an active research area [2, 9, 10, 12, 19].

While lookup tables can hold any arbitrary data, the primary emphasis in this paper revolves around their applications in function evaluation. Using lookup tables for function evaluation is an efficient method due to its simplicity of implementation, low computational latency, and high throughput, especially for evaluating compound complex functions, such as $1/[1 + exp(-x)]$, which can be evaluated by a table of precomputed values in hardware instead of performing costly intermediate operations step-by-step.

At low resolutions, lookup tables can be directly used for implementing a function by tabulating the values of all possible inputs. Given a function at the input resolution w_{in} and output resolution w_{out}, the size of the corresponding lookup table would be $w_{out} \times 2^{w_{in}}$ bits, which grows exponentially as w_{in} increases. For this reason, this approach is usually used for evaluating a function at up to 12-bit resolutions [16].

At higher resolutions, however, simple tabulation of a function is not feasible due to the massive sizes of resulting tables. In such cases, approximate methods are applied, which sacrifice accuracy for hardware cost savings. Examples of such methods include bipartite table (BT) [18], multipartite table (MT) [4, 11], and piecewise polynomial approximation (PPA) [5] methods. BT and MT decompose the table of a function into smaller tables, called the table of initial values (TIV) and table of offsets (TO), which result in the reduction of hardware costs. PPA methods, however, break a

function into sub-functions and approximate them with polynomials whose coefficients are stored in smaller tables. Although all of these methods can simplify the implementation of a function at the expense of accuracy, they still rely on lookup tables to store essential values such as TIV, TO, or tables of coefficients. Lookup tables are also used in other state-of-the-art methods, libraries, and architectures for high-resolution function evaluation. For instance, hls4ml [6, 8] is a Python package for machine learning interface on FPGAs, and it uses lookup tables to perform nonlinear parts of activation functions in neural networks. Additionally, many floating-point operations require lookup tables as parts of their architectures [1, 15, 17].

As a result, lookup tables are used either directly for function evaluation or as parts of other table-based methods. In either case, table compression methods can be used to shrink such tables to reduce their implementation hardware costs.

In this paper, we propose CompressedLUT as a method for lossless compression of lookup tables, which uses the idea of decomposition [2, 12], self-similarities [13, 14], multilevel compression, and higher-bit compression to maximize table size savings. CompressedLUT is available as an open source tool[1].

To evaluate our method, we compressed and implemented several lookup tables on FPGAs using HLS to show the efficiency of CompressedLUT against the PlainTable (conventional plain table) approach and previous compression methods, including TwoTable (two-table decomposition) [12] and LDTC (lossless differential table compression) [2]. Some of the lookup tables were obtained by directly tabulating the values of 12-bit elementary functions, and some others were the lookup tables used by other table-based methods and libraries for approximating functions at higher resolutions, such as the MT method at 24-bit, PPA method at 36-bit, and hls4ml library at 18-bit resolutions. Our CompressedLUT achieved a compression ratio of 60%, while TwoTable and LDTC achieved compression ratios of 33% and 37% on average, respectively. In terms of throughput per slice hardware cost, our method resulted in 2.33 times higher throughput than PlainTable, while TwoTable and LDTC resulted in 1.63 and 1.29 times higher throughput than PlainTable on average, respectively.

The rest of the paper is as follows. Section 2 discusses the details of each technique used for compression. In Section 3, the implementation results are presented and discussed. Finally, the paper is concluded in Section 4.

2 METHODOLOGY

We describe our compression methodology by first presenting the idea of breaking a table into two smaller tables (Sec. 2.1), similar to what TwoTable [12] and LDTC [2] use. Then we use the idea of finding self-similarities in the smaller table (Sec. 2.2), extending the idea in [14].

The above methods would be suitable for tables that store functions that are smooth and have small local variations. However, for tables that store values with higher dynamic range and large local variations (such as the ones used in many function approximation methods), we present two other techniques detailed in sections 2.3 and 2.4. The overall architecture of our method is discussed in Sec. 2.5.

[1]CompressedLUT is available at **https://github.com/kiabuzz/CompressedLUT** (DOI: 10.5281/zenodo.10431619).

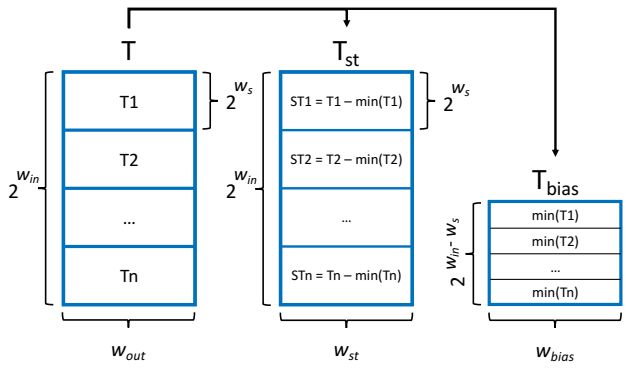

Figure 1: Decomposition of T into T_{bias} and T_{st}.

2.1 Lookup Table Decomposition

Similar to TwoTable [12] and LDTC [2], we decompose a table T into two new tables T_{bias} and T_{st}. Fig 1 shows the decomposition of T into T_{bias} and T_{st}. Assuming T has $2^{w_{in}}$ elements of w_{out} bits, it is split into $n = 2^{w_{in}-w_s}$ sub-tables, where $0 < w_s < w_{in}$. Next, the minimum value of each sub-table is stored as an element in T_{bias}. Additionally, the minimum value of each sub-table is subtracted from all the values in the corresponding sub-table and the resulting values are stored in T_{st}.

As seen, T_{bias} has $2^{w_{in}-w_s}$ elements of w_{bias} bits, where w_{bias} is usually the same as w_{out}. Whereas T_{st} has $2^{w_{in}}$ elements of w_{st} bits, where w_{st} is less than w_{out}. This is because T_{st} holds local variations which usually require a smaller bit width. In summary, the table T_{bias} has the same output bit width as the original table T, but it has fewer elements. In contrast, the table T_{st} has the same number of elements as the original table T, but it has less output bit width. The tables have the following number of bits.

$$Size(T) = 2^{w_{in}} \times w_{out}$$
$$Size(T_{bias}) = 2^{w_{in}-w_s} \times w_{bias}$$
$$Size(T_{st}) = 2^{w_{in}} \times w_{st}$$

The final size ratio obtained by table decomposition is as follows.

$$\begin{aligned} SizeRatio &= [Size(T_{bias}) + Size(T_{st})]/Size(T) \\ &= [2^{w_{in}-w_s} \times w_{bias} + 2^{w_{in}} \times w_{st}]/(2^{w_{in}} \times w_{out}) \\ &= 2^{-w_s} + w_{st}/w_{out} \end{aligned}$$

As seen, the final size ratio after decomposition depends on two terms: 2^{-w_s} and w_{st}/w_{out}. The parameter w_s can be set to any value between 0 and w_{in}. Increasing w_s decreases the first term 2^{-w_s}, yet it increases the second term w_{st}/w_{out}. This is because increasing w_s results in sub-tables with more elements, which might have larger local variations, that require greater bit width w_{st}.

After decomposition, the original table T is replaced by T_{bias} and T_{st}. The input address of T_{st} is the same as the input address of T, but the input address of T_{bias} is fed by the $(w_{in} - w_s)$ higher bits of the input address of T. Finally, an adder is used to retrieve

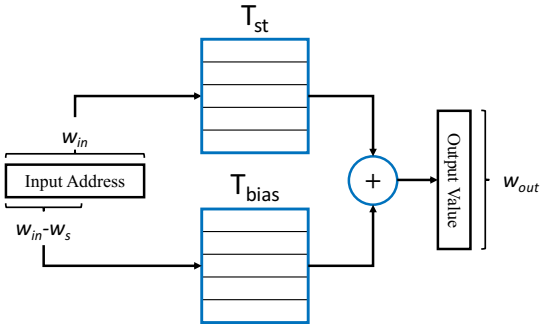

Figure 2: Retrieving T through T_{bias} and T_{st}.

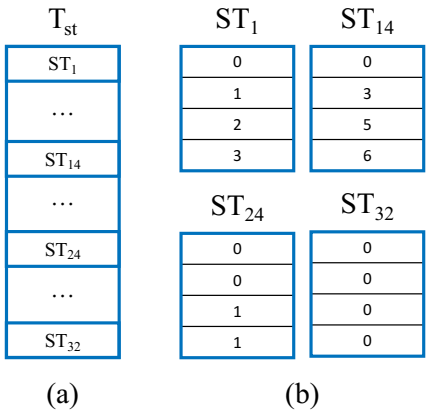

Figure 3: Examples of self-similarities in sub-tables.

the values of the original table T by adding the output values of T_{st} and T_{bias}, as seen in Fig 2.

2.2 Self-Similarities in Lookup Tables

Using the core idea of what the authors of [14] call the "SimBU" method, we can compress the table of T_{st} further. SimBU was proposed in the context of "unary" methods to reduce the complexity of HBU [7]. However, the self-similarity algorithm proposed by this method can be deployed as a lossless compression approach in the context of binary lookup tables.

As discussed in Section 2.1, T_{st} holds the values of n sub-tables ST_i, where $i \in \{1, 2, \cdots, n\}$. However, the values of many of these sub-tables are similar. Similar sub-tables refer to the sub-tables whose values are either identical or can get identical through the arithmetic right shift operation.

Fig 3a shows an example of T_{st} which contains 32 sub-tables of 4 elements. Therefore, T_{st} has 128 elements in total. The values of four sub-tables ST_1, ST_{14}, ST_{24}, and ST_{32} are shown separately in Fig. 3b. As seen, if the values of ST_{14} are shifted to the right by 1 bit, we can obtain ST_1. Additionally, if the values of ST_{14} are shifted to the right by 2 or 3 bits, we can obtain ST_{24} or ST_{32}, respectively. In other words, we can say that ST_{14} can generate ST_1, ST_{24}, and ST_{32} using the right shift operation. As a result, instead of storing 4 different sub-tables, we can store only ST_{14} as a unique sub-table, through which we can retrieve the other ones.

In the example of Fig. 3b, in addition to ST_{14}, ST_1 can also generate ST_{24} and ST_{32}, but this time by getting shifted to the right by 1 and 2 bits, respectively. Furthermore, ST_{24} can generate ST_{32} by getting shifted to the right by 1 bit. However, among these 4 sub-functions, considering ST_{14} as the unique sub-function is the best choice since it can generate 3 other sub-functions. Therefore, the final goal of this phase is to find the minimum set of unique sub-tables in T_{st} that can generate the rest.

Using the self-similarity matrix used in SimBU [14], similarities among all sub-tables in T_{st} are identified. As discussed in Section 2.1, T_{st} consists of $n = 2^{w_{in}-w_s}$ sub-tables, and each sub-table consists of 2^{w_s} elements. To measure similarities, an $n \times n$ Boolean matrix is needed, which is called a similarity matrix. Each entry of this matrix specifies whether the two sub-tables are similar or not. That is, an entry s_{ij} is 1 if the sub-table ST_i can generate ST_j. Obviously, this matrix is not symmetric since if ST_i can generate ST_j through right shifting, the opposite is not necessarily true. The following is the definition of the similarity matrix.

$$SimilarityMatrix = \begin{bmatrix} sm_{1,1} & sm_{1,2} & \cdots & sm_{1,n} \\ sm_{2,1} & sm_{2,2} & \cdots & sm_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ sm_{n,1} & sm_{n,2} & \cdots & sm_{n,n} \end{bmatrix}_{n \times n} \quad (1)$$

$$sm_{i,j} = 1 \Leftrightarrow \exists t \in \mathbb{N} : \forall m \in \mathbb{N}(m < 2^{w_s}), rsh_t\{ST_i[m]\} = ST_j[m]$$

where rsh_t denotes an arithmetic right shift by t bits.

After identifying similar sub-tables, the unique set of them should be determined that can generate the other sub-tables to retrieve the original T_{st}. Unique sub-tables are named UST, and they are all stored in a new single table, called T_{ust}. Furthermore, two new tables of n elements, called T_{idx} and T_{rsh}, are needed to retrieve the original table T_{st} through T_{ust}. The value of the ith element in T_{idx} shows the index of the unique sub-function that can generate ST_i, and the value of the ith element in T_{rsh} shows the number of right bit shifts that are needed to be performed on the values of the corresponding unique sub-table to retrieve ST_i. For instance, if $T_{idx}[5] = 3$ and $T_{rsh}[5] = 2$, we can conclude that ST_5 can be retrieved by UST_3 after right shifting the values of UST_3 by 2 bits.

To find unique sub-tables, a vector must be obtained based on the similarity matrix. This vector is called a similarity vector, and the jth entry in it specifies how many sub-tables can be generated using the jth sub-table. The vector can be created by adding the values in each column in the similarity matrix as follows.

$$SimilarityVector = [sv_1, sv_2, \cdots, sv_n]$$
$$sv_j = \sum_i sm_{ij} \quad (2)$$

The index of the element in the similarity vector with the maximum value determines the first unique sub-table. In other words, if sv_i is the element with the maximum value, ST_i will be considered as the first unique sub-table UST_1, and its values are stored in T_{ust}. We also need to traverse through the ith column of the similarity matrix to see which sub-tables can be generated through ST_i. If ST_i can generate ST_j through right shifting by t bits, then the jth element of T_{idx} and T_{rsh} must be set to 1 and t, respectively. After finding the first unique sub-table, we need to update the similarity matrix and similarity vector. Therefore, the ith row and column

of the similarity matrix must be set to 0. Additionally, if ST_i can generate ST_j, the jth row and column of the similarity matrix must be set to 0 as well. The elements of the similarity vector need to be recalculated based on the updated similarity matrix.

The process above needs to be repeated again and again until all the entries of the similarity matrix are 0's. In each iteration, it identifies a new unique sub-table. In the end, if the process takes k iterations to finish, we will end up with k unique sub-tables UST_i, where $i \in \{1, 2, \cdots, k\}$ and $k \leq n$. These unique sub-tables are all stored in T_{ust}.

As a result, T_{st} is replaced by T_{ust}, T_{idx}, and T_{rsh}. In contrast to T_{st}, which contains n sub-tables, T_{ust} contains k unique sub-tables, where k is often *by far* less than n. It means that many sub-tables can be generated using a few unique sub-tables. Therefore, we can achieve significant table size reductions. However, when calculating the overall memory space reduction, the size of T_{idx} and T_{rsh} must be taken into account. In summary, the size of each table and the size ratio are as follows.

$$Size(T_{st}) = n \times 2^{w_s} \times w_{st}$$
$$Size(T_{ust}) = k \times 2^{w_s} \times w_{st}$$
$$Size(T_{idx}) = n \times w_{idx}$$
$$Size(T_{rsh}) = n \times w_{rsh}$$

$$\begin{aligned} SizeRation &= [Size(T_{ust}) + Size(T_{idx}) + Size(T_{rsh})]/Size(T_{st}) \\ &= [w_{idx} + w_{rsh}]/(2^{w_s} \times w_{st}) + k/n \end{aligned}$$

where w_{idx} and w_{rsh} are the bit width of the values in T_{idx} and T_{rsh}, respectively. In our method, however, we force w_{rsh} to be 2, which means that during the self-similarity search process, we limit the value of t in Eq. 1 to the range of [0,3]. The value of w_{idx} depends on the number of unique sub-tables and is equal to $floor(log2(k-1)) + 1$.

2.3 Higher-Bit Compression

Using decomposition and self-similarities can potentially reduce a table's size, especially if the values of a table change continuously. That is, these two compression techniques can be more efficient if there are small differences between consecutive values in a table. On the other hand, there are two issues in the compression of tables with more discrete values that show large differences between consecutive elements.

The first issue is the increase of w_{st} in T_{st} after decomposition (Section 2.1), which negatively impacts the final table size savings. This is because there are larger differences between consecutive values in T, and therefore the local variations are higher. As a result, the values in T_{st}, which stores the local variations, require a longer bit width w_{st}. The second issue is with self-similarities (Section 2.2). Since the values of sub-tables are larger, it is likely harder to find similarities among them. Therefore, the number of unique sub-tables increases, which in turn results in lower table size savings.

As a solution to mitigate these issues, we can split the values of T into higher and lower bits before performing decomposition and self-similarity measures. The values of T are divided into w_l lower bits and $w_{out} - w_l$ higher bits, which can be stored in two separate tables T_{lb} and T_{hb}, respectively. The table T_{lb} undergoes no compression, but T_{hb} is compressed by using decomposition (Section 2.1) and self-similarities (Section 2.2).

The intuition behind this practice is to reduce the distances between consecutive values of T by considering higher bits. If we plot both T and T_{hb}, the overall shapes of the plots will be similar, however, the slopes of sub-regions in the plot of T_{hb} would be more gentle. Therefore, local variations become lower, which potentially results in more table size savings after using decomposition and self-similarity techniques.

2.4 Multilevel Compression

Using the three techniques discussed in Sections 2.1, 2.2, and 2.3, a table T can be significantly compressed and replaced by T_{lb}, T_{ust}, T_{idx}, T_{rsh}, and T_{bias}. Among these tables, T_{bias} can be compressed further by performing all three techniques on it. As a result, T_{bias} itself is replaced by another set of T_{lb}, T_{ust}, T_{idx}, T_{rsh}, and T_{bias}. This can potentially achieve further table size savings in total.

It is worth noting that if we plot the values of T and T_{bias}, they will have a similar shape. This is because T_{bias} is the same as T sampled by a factor of $2^{w_{in}-w_s}$. Although T_{bias} has a coarser granularity than T, this issue can be resolved by splitting the values of T_{bias} into higher and lower bits, as discussed in Section 2.3.

Using the idea of multilevel compression often results in more table size savings. However, it might increase hardware costs due to the nested decoders needed to retrieve values.

2.5 Overall Architecture

Algorithm 1 describes the compression techniques used by our CompressedLUT method. This algorithm takes a table T and two parameters w_s and w_l as inputs, and it returns five tables T_{lb}, T_{ust}, T_{bias}, T_{idx}, and T_{rsh} as outputs. For multilevel compression, the algorithm must be run again multiple times, given T_{bias} as input. Fig. 4 shows the overall architecture of our method.

The parameters w_s and w_l should be determined for each specific input table T. In our method, we run the algorithm for different values of the parameters and evaluate them based on the total sizes of all generated tables. Although the runtime of this procedure highly depends on the initial size of a lookup table, our CompressedLUT tool takes around 1.38 seconds on a regular computer to compress a lookup table of 4096 values at 12-bit resolution.

3 IMPLEMENTATION RESULTS

In this section, we evaluate the efficiency of CompressedLUT by implementing a number of tables of different sizes and comparing our method with the previous TwoTable [12] and LDTC [2] works. Fig. 5 show the graphs of implemented lookup tables.

In section 3.1, we present and discuss the results of implementing tables, which are used directly for function evaluation at low resolutions. In Section 3.2, however, we present and discuss the results of implementing tables, which are used by other methods for approximate function evaluation at higher resolutions. In Section 3.3, we finally discuss the potential applications of our method to general compression applications, such as audio and image data compression.

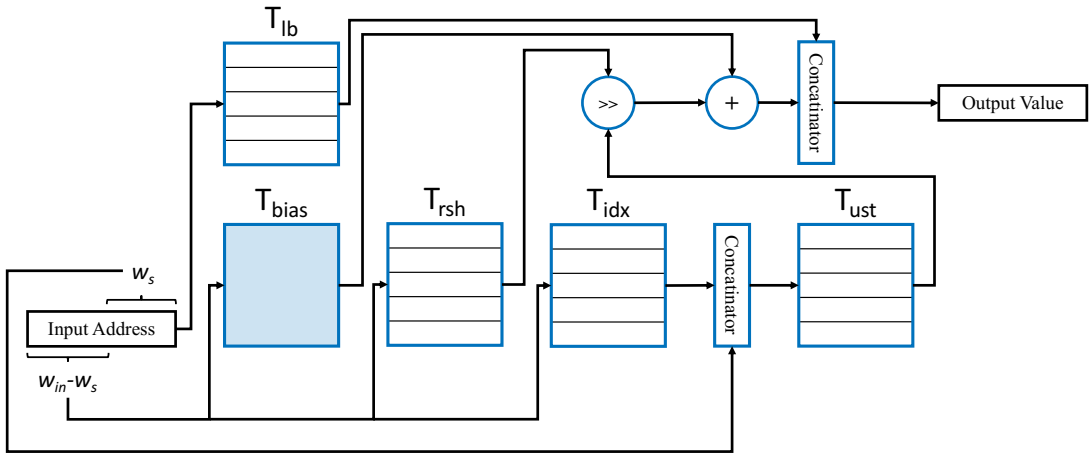

Figure 4: Overall architecture of our CompressedLUT method. In the case of multilevel compression, the same architecture is embedded in T_{bias}.

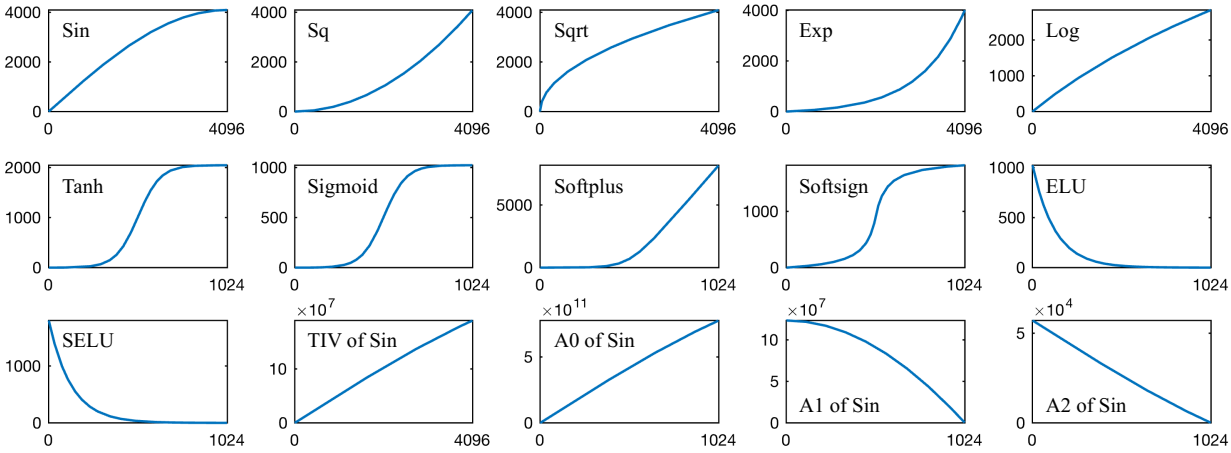

Figure 5: Plots of the implemented lookup tables. X-axes represent the input address of the tables, and Y-axes represent the corresponding output values. *Note the different scales on the x-axis of different plots.*

3.1 Low-Resolution Function Evaluation

As discussed earlier, a low-resolution function at up to 12-bit can be evaluated directly by lookup tables containing the values of the function for all possible input values [2].

Such tables can be compressed using our lossless compression method, which can reduce hardware costs with no accuracy loss. To show the efficiency of our method for low-resolution function evaluation, we targeted a number of nonlinear functions at 12-bit resolution, each of which had a table of $12 \times 2^{12} = 49152$ bits. The minimum value of each table is subtracted from all the values in that table, which could potentially remove excessive output bits such as sign bits in some cases. The first row in Fig. 5 shows the plots of implemented tables for low-resolution function evaluation. We used our CompressedLUT method as well as previous TwoTable [12] and LDTC [2] works to compress the tables as much as possible. As in [12], we used total bit count as a metric to guide the selection of decomposition parameters in each method. Finally, we synthesized and implemented the compressed tables and their decoders on AMD's Kintex-7 FPGAs using Vitis HLS 2023.1.

We obtained hardware utilization and timing reports after place and route. Average latency and throughput were measured after cosimulation. Each table in HLS was implemented as a single-port ROM (ROM_1P) using distributed RAM (LUTRAM) resources instead of block RAM (BRAM) resources, as using BRAMs cannot show the efficiency of the compression methods due to the discrete sizes of BRAMs [2]. Therefore, we can consider the number of utilized slices as a metric for area measurements since the designs do not use other hardware resources such as BRAMs and DSP blocks. However, we use throughput per slice (TPS) as a metric to compare different methods in terms of hardware cost. The latency of our method is usually 2 clock cycles, whereas the latency of the PlainTable, TwoTable, and LDTC methods is 1. Nonetheless, all

Algorithm 1: CompressedLUT

Input: T, w_l, w_s
Outputs: $T_{lb}, T_{ust}, T_{bias}, T_{idx}, T_{rsh}$
$T_{lb}[:] \leftarrow bitand(T[:], 2^{w_{lb}} - 1)$
$T_{hb}[:] \leftarrow rsh(T[:], w_{lb})$
$w_{in} \leftarrow bitwidth(length(T) - 1)$
$w_{out} \leftarrow bitwidth(max(T))$
$n \leftarrow 2^{w_{in} - w_s}$
# Compression of T_{hb} Using Decomposition
for $i = 1$ to n **do**
  $ST[:] \leftarrow T_{hb}[(i-1) \times 2^{w_s} + 1 : i \times 2^{w_s}]$
  $T_{st}[(i-1) \times 2^{w_s} + 1 : i \times 2^{w_s}] \leftarrow ST[:] - min(ST[:])$
  $T_{bias}[i] \leftarrow min(ST[:])$
end
Compression of T_{st} Using Self-Similarities
$SimilarityMatrix[:][:] \leftarrow zeros(n, n)$
$RighShiftMatrix[:][:] \leftarrow zeros(n, n)$
for $i = 1$ to n **do**
  $ST_i \leftarrow T_{st}[(i-1) \times 2^{w_s} + 1 : i \times 2^{w_s}]$
  for $j = 1$ to n **do**
    $ST_j \leftarrow T_{st}[(j-1) \times 2^{w_s} + 1 : j \times 2^{w_s}]$
    for $t = 0$ to 3 **do**
      if $rsh(ST_i[:], t) == ST_j[:]$ **then**
        $SimilarityMatrix[i][j] \leftarrow 1$
        $RighShiftMatrix[i][j] \leftarrow t$
        break
      end
    end
  end
end
$k \leftarrow 0$
$SimilarityVector[:] \leftarrow zeros(1, n)$
while $SimilarityMatrix[:][:] != zeros(n, n)$ **do**
  $k \leftarrow k + 1$ # increment the number of unique sub-tables
  $SimilarityVector[:] \leftarrow \sum_i SimilarityMatrix[i][:]$
  $idx \leftarrow \arg\max_i SimilarityVector[i]$
  $UST[:] \leftarrow T_{st}[(idx-1) \times 2^{w_s} + 1 : idx \times 2^{w_s}]$
  $T_{ust}[(k-1) \times 2^{w_s} + 1 : k \times 2^{w_s}] \leftarrow UST[:]$
  $T_{idx}[idx] \leftarrow k$
  $SimilarityMatrix[idx][idx] \leftarrow 0$
  # reference similar sub-tables to the kth unique sub-table
  for $i = 1$ to n **do**
    if $SimilarityMatrix[i][idx] == 1$ **then**
      $T_{idx}[i] \leftarrow k$
      $T_{rsh} \leftarrow RighShiftMatrix[i][idx]$
      $SimilarityMatrix[i][:] \leftarrow zeros(1, n)$
      $SimilarityMatrix[:][i] \leftarrow zeros(n, 1)$
    end
  end
  $SimilarityMatrix[idx][:] \leftarrow zeros(1, n)$
end

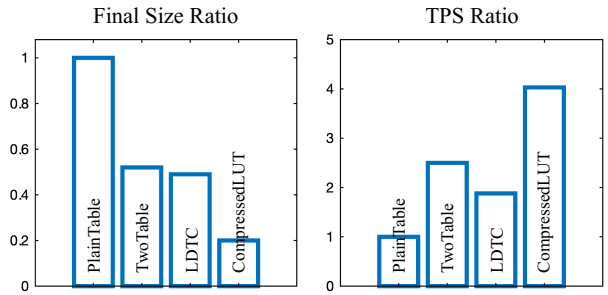

Figure 6: Average of the results of implementing lookup tables used directly for low-resolution function evaluation.

the designs were pipelined with an initial interval (II) of 1 clock cycle, which means that all the designs would accept a new input value every clock cycle. To make comparisons fair, we set the minimum latency of all the methods to 2 clock cycles. As expected, this constraint improved the average TPS values in the PlainTable, TwoTable, and LDTC methods due to reductions in critical path delay, which resulted in an increase in throughput.

Table 1 shows the results of implementing the lookup tables using different methods. Table 2 and Fig. 6 also show the average of the results for each method. PlainTable is referred to as a method using simple plain tables with no compression. "Initial Size" and "Final Size" show the total bit count before and after compression, respectively. "Delay" shows the achieved clock period in nanoseconds (ns), and "TPS" shows throughput per slice in mega operations per second per slice (Mops/slice).

As seen, our method can compress the tables on average by 80%, whereas the TwoTable and LDTC methods compress them on average by 48% and 51%, respectively. In terms of the TPS hardware cost, our method is 4.03 times better than the PlainTable method, whereas the TwoTable and LDTC methods are 2.50 and 1.88 times better than the PlainTable method, respectively.

3.2 High-Resolution Function Evaluation

Unlike low-resolution functions, it is not practical to fully tabulate the values of a function beyond 12-bit due to the exponentially growing size of the resulting tables. In such cases, approximate methods, such as BT, MT, and PPA can be applied to reduce overall table size at the expense of accuracy. As discussed earlier, these approximate methods still rely on lookup tables to store essential values to perform computations. For instance, BT and MT methods rely on TIV and TO tables. In addition, PPA methods store the coefficients of polynomials in lookup tables.

Our CompressedLUT method can be plugged into such table-based methods to compress their tables, which reduces hardware costs with no additional approximation error. However, compressing such tables is not as easy as compressing the tables of low-resolution functions, discussed in Section 3.1. This is because the lookup tables used in table-based methods usually do not show smooth local variations compared to the tables of low-resolution functions. For instance, the TIV table of a function, implemented by an MT method, contains uniformly sampled values of the function. Therefore, the difference between every two consecutive values in

Table 1: Results of implementing lookup tables used directly for low-resolution function evaluation on FPGA using HLS. The costs of decoders are included, and all the designs are pipelined with II = 1.

Specifications			Compression		Hardware Costs					
Function	Initial Size (bit)	Method	Final Size (bit)	Ratio	Slice	LUT	FF	Delay (ns)	TPS (Mops/slice)	Ratio
Sin	49152	Plain Table	49152	1.00	180	605	70	3.453	1.61	1.00
		TwoTable [12]	22528	0.46	65	238	43	3.588	4.29	2.67
		LDTC [2]	21504	0.44	90	322	41	3.42	3.25	2.02
		CompressedLUT	8764	0.18	53	165	82	2.808	6.72	4.18
Sq	49152	Plain Table	49152	1.00	158	543	51	3.197	1.98	1.00
		TwoTable [12]	22528	0.46	50	182	43	3.067	6.52	3.29
		LDTC [2]	22016	0.45	65	223	42	3.841	4.01	2.02
		CompressedLUT	9472	0.19	48	148	68	2.667	7.81	3.95
Sqrt	49152	Plain Table	49152	1.00	132	462	53	3.67	2.06	1.00
		TwoTable [12]	38400	0.78	98	350	48	4.002	2.55	1.24
		LDTC [2]	33792	0.69	149	493	39	4.158	1.61	0.78
		CompressedLUT	13696	0.28	59	208	60	3.278	5.17	2.50
Exp	49152	Plain Table	49152	1.00	177	638	73	4.221	1.34	1.00
		TwoTable [12]	26624	0.54	68	237	44	3.661	4.02	3.00
		LDTC [2]	25600	0.52	87	312	42	3.767	3.05	2.28
		CompressedLUT	11392	0.23	61	203	75	2.846	5.76	4.30
Log	49152	Plain Table	49152	1.00	163	605	70	3.283	1.87	1.00
		TwoTable [12]	18432	0.38	66	230	42	3.493	4.34	2.32
		LDTC [2]	18432	0.38	66	220	42	3.528	4.29	2.30
		CompressedLUT	5924	0.12	40	124	74	2.573	9.72	5.20

Table 2: Average of the results of implementing lookup tables used directly for low-resolution function evaluation.

Method	Final Size Ratio	TPS Ratio
PlainTable	1.00	1.00
TwoTable [12]	0.52	2.50
LDTC [2]	0.49	1.88
CompressedLUT	0.20	4.03

the TIV table is likely larger than that of two consecutive values in the plain table of the function. Nonetheless, our method can achieve significant savings in table size due to breaking output values into higher bits and lower bits as well as using a multilevel compression technique. To show the effectiveness of our method for high-resolution function evaluation, we implemented the tables used by the following methods and libraries.

(1) hls4ml is a Python package for machine learning interface on FPGAs using HLS. It uses lookup tables to implement the nonlinear parts of activation functions in neural networks. For linear parts, however, it might use compactors and constant-coefficient multipliers. By default, the lookup tables have 1024 elements of fixed point values with 10 fractional bits and 8 signed integer bits (ap_fixed<18, 8>). However, the integer parts of the values hardly exceed 2 bits in most cases. We implemented a number of activation functions using hls4ml [6, 8] at 18-bit resolution, including tanh, sigmoid, softplus, softsign, ELU (exponential linear unit), SELU (scaled exponential linear unit). Next, we compressed and implemented the lookup tables used by the architecture of each activation function.
(2) FloPoCo [3] is an arithmetic core generator for FPGAs. It includes several tools for approximating arbitrary functions. It has a tool, called FixFunctionByMultipartiteTable for function evaluation using an MT method [4]. In this paper, this tool is referred to as FloPoCo-MT (multipartite table method). We generated the arithmetic core for $sin(\frac{\pi}{4}x)$ at 24-bit resolution using FloPoCo-MT. Next, we compressed and implemented the TIV (table of initial values) used by the MT method in the generated core.
(3) FloPoCo [3] has another tool, called FixFunctionByPiecewisePoly, for function evaluation using a PPA method [5] based on the Horner scheme. In this paper, this tool is referred to as FloPoCo-PPA (piecewise polynomial approximation method). We generated the arithmetic core for $sin(\frac{\pi}{4}x)$ at 36-bit resolution using FloPoCo-PPA with quadratic polynomials ($a_2x^2 + a_1x + a0$). Next, we separately compressed and implemented the tables of coefficients, referred to as A0, A1, and A2.

The minimum value of each table was subtracted from all the values in that table to potentially remove excessive output bits. The second

Table 3: Results of implementing lookup tables used by other table-based methods for high-resolution function evaluation on FPGA using HLS. The costs of decoders are included, and all the designs are pipelined with II = 1.

Specifications				Compression		Hardware Costs					
Architecture	Table	Initial Size (bit)	Method	Final Size (bit)	Ratio	Slice	LUT	FF	Delay (ns)	TPS (Mops/slice)	Ratio
hls4ml [6, 8] (18-bit)	Tanh	11264	Plain Table	11264	1.00	45	143	34	2.98	7.46	1.00
			TwoTable [12]	7552	0.67	30	108	41	2.62	12.72	1.71
			LDTC [2]	7168	0.64	31	112	37	2.86	11.28	1.51
			CompressedLUT	4976	0.44	30	97	54	2.15	15.53	2.08
	Sogmoid	10240	Plain Table	10240	1.00	33	103	33	2.30	13.18	1.00
			TwoTable [12]	6400	0.63	25	88	38	2.59	15.46	1.17
			LDTC [2]	6144	0.60	25	94	35	2.48	16.15	1.23
			CompressedLUT	3984	0.39	28	81	51	2.02	17.65	1.34
	Softplus	13312	Plain Table	13312	1.00	38	133	36	2.19	12.02	1.00
			TwoTable [12]	8832	0.66	33	113	46	3.02	10.03	0.84
			LDTC [2]	8320	0.63	35	116	42	2.70	10.59	0.88
			CompressedLUT	5664	0.43	39	124	70	2.32	11.08	0.92
	Softsign	11264	Plain Table	11264	1.00	46	157	34	2.92	7.45	1.00
			TwoTable [12]	8576	0.76	32	116	42	2.78	11.23	1.51
			LDTC [2]	7808	0.69	59	182	36	3.54	4.79	0.64
			CompressedLUT	4636	0.41	33	101	68	2.37	12.78	1.71
	ELU	11264	Plain Table	11264	1.00	30	98	34	2.16	15.46	1.00
			TwoTable [12]	7808	0.69	28	94	41	2.65	13.46	0.87
			LDTC [2]	6912	0.61	28	92	36	2.59	13.78	0.89
			CompressedLUT	4232	0.38	28	84	50	2.42	14.75	0.95
	SELU	11264	Plain Table	11264	1.00	32	113	34	2.53	12.36	1.00
			TwoTable [12]	8576	0.76	29	97	42	2.87	12.00	0.97
			LDTC [2]	7936	0.70	30	105	37	2.94	11.33	0.92
			CompressedLUT	5008	0.44	27	87	55	2.03	18.26	1.48
FloPoCo-MT [3, 4] (24-bit)	TIV of Sin	114688	Plain Table	114688	1.00	470	1672	150	4.00	0.53	1.00
			TwoTable [12]	89088	0.78	326	1192	145	4.06	0.76	1.42
			LDTC [2]	83456	0.73	465	1580	150	4.87	0.44	0.83
			CompressedLUT	64724	0.56	345	1136	176	3.60	0.80	1.51
FloPoCo-PPA [3, 5] (36-bit)	A0 of Sin	40960	Plain Table	40960	1.00	191	692	89	3.57	1.47	1.00
			TwoTable [12]	37120	0.91	191	668	116	3.70	1.42	0.97
			LDTC [2]	34944	0.85	180	642	87	3.22	1.73	1.18
			CompressedLUT	32184	0.79	172	545	107	3.08	1.89	1.29
	A1 of Sin	27648	Plain Table	27648	1.00	132	458	62	3.72	2.03	1.00
			TwoTable [12]	24256	0.88	125	434	80	3.73	2.15	1.06
			LDTC [2]	22528	0.81	115	418	64	3.28	2.65	1.30
			CompressedLUT	19704	0.71	97	349	92	2.67	3.86	1.90
	A2 of Sin	16384	Plain Table	16384	1.00	69	242	45	2.83	5.12	1.00
			TwoTable [12]	11264	0.69	44	148	54	2.96	7.69	1.50
			LDTC [2]	10496	0.64	87	265	48	3.70	3.11	0.61
			CompressedLUT	8120	0.50	43	145	57	2.82	8.25	1.61

and the third rows in Fig. 5 show the plots of implemented tables used by the aforementioned methods for high-resolution function evaluation. Compression was made by our CompreesedLUT method as well as previous TwoTable [12] and LDTC [2] methods. We used the total bit count as the metric to guide the selection of decomposition parameters in each method. All the designs were synthesized and implemented on AMD's Kintex-7 FPGAs using Vitis HLS 2023.1. All the considerations for developing HLS designs and obtaining results are the same as those in Section 3.1.

Table 3 shows the implementation results using different methods. Additionally, Table 4 and Fig. 7 show the average results for each method. As seen, our CompressedLUT can compress the tables on average by 50%, which results in 1.48 times better TPS than the PlainTable method. In comparison, TwoTable and LDTC compress the tables on average by 26% and 31%, respectively. In terms of TPS, TwoTable is 1.20 times better than PlainTable, but LDTC has the same TPS as PlainTable.

Table 4: Average of the results of implementing lookup tables used by other table-based methods for high-resolution function evaluation.

Method	Final Size Ratio	TPS Ratio
PlainTable	1.00	1.00
TwoTable [12]	0.74	1.20
LDTC [2]	0.69	1.00
CompressedLUT	0.50	1.48

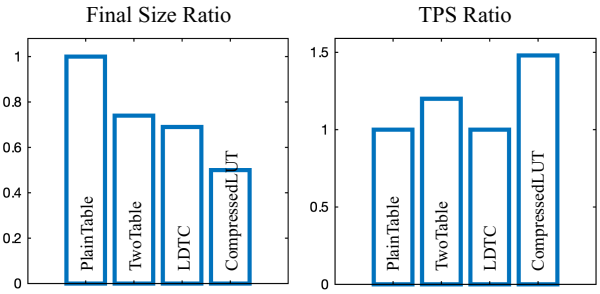

Figure 7: Average of the results of implementing lookup tables used by other table-based methods for high-resolution function evaluation.

3.3 Beyond Function Evaluation

As observed, CompressedLUT can significantly compress lookup tables used for function evaluation. However, our method is not limited to math functions and can be used as a general lossless compression scheme to store data either on-chip or off-chip using less memory resources. The benefit of our method is that while it compresses data, it does not require complex and costly decoders to decompress data in hardware. In other words, original data can be recovered using hardware-efficient decoders, that can be easily pipelined to maximize throughput.

Moreover, our method can be extended to multidimensional lookup tables. For instance, a two-dimensional (2D) table can be compressed by decomposing it into smaller 2D sub-tables and finding unique sets of them that can recover the original table through simple transformations. Multilevel compression and higher-bit compression techniques can be applied as well. Therefore, some sort of data, such as image data, might be more effectively compressed using CompressedLUT for multidimensional lookup tables. We are planning to address these problems in our future work.

4 CONCLUSIONS

In this paper, CompressedLUT was proposed as a lossless lookup table compression method, which uses multilevel compression, decomposition, self-similarities, and other techniques to compress arbitrary arrays of data, implemented as lookup tables. We showed the effectiveness of our method by implementing a number of lookup tables, which were either used directly for direct 12-bit function evaluation or used by other approximate methods and libraries for function evaluation at higher resolutions, such as the MT (multipartite table) method at 24-bit, PPA (piecewise polynomial approximation) method at 36-bit, and hls4ml activation functions at 18-bit resolutions. Our method compressed the lookup tables on average by 60%, while previous TwoTable and LDTC compressed them on average by 33% and 37%, respectively. In terms of throughput per slice hardware cost, our method was on average 2.33 times better than conventional implementations, while TwoTable and LDTC were 1.63 and 1.29 times better than the conventional implementations. Although this paper focused on the compression of lookup tables used for function evaluation, our method can be applied to any array of data.

ACKNOWLEDGMENTS

This material is based upon work supported in part by Cisco Systems, Inc. under grant number 00105407, and by the National Science Foundation under grant number PFI-TT 2016390.

REFERENCES

[1] Nelson Campos, Slava Chesnokov, Eran Edirisinghe, and Alexis Lluis. 2021. FPGA Implementation of Custom Floating-Point Logarithm and Division. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, Steven Derrien, Frank Hannig, Pedro C. Diniz, and Daniel Chillet (Eds.). Springer International Publishing, Cham, 295–304.

[2] Maxime Christ, Luc Forget, and Florent de Dinechin. 2022. Lossless Differential Table Compression for Hardware Function Evaluation. *IEEE Transactions on Circuits and Systems II: Express Briefs* 69, 3 (2022), 1642–1646. https://doi.org/10.1109/TCSII.2021.3131405

[3] Florent de Dinechin and Bogdan Pasca. 2011. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers* 28, 4 (2011), 18–27. https://doi.org/10.1109/MDT.2011.44

[4] F. de Dinechin and A. Tisserand. 2005. Multipartite table methods. *IEEE Trans. Comput.* 54, 3 (2005), 319–330. https://doi.org/10.1109/TC.2005.54

[5] J. Detrey and F. de Dinechin. 2005. Table-based polynomials for fast hardware function evaluation. In *2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*. 328–333. https://doi.org/10.1109/ASAP.2005.61

[6] Javier Duarte et al. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *JINST* 13, 07 (2018), P07027. https://doi.org/10.1088/1748-0221/13/07/P07027 arXiv:1804.06913 [physics.ins-det]

[7] S. Rasoul Faraji and Kia Bazargan. 2019. Hybrid Binary-Unary Hardware Accelerator. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference* (Tokyo, Japan) *(ASPDAC '19)*. Association for Computing Machinery, New York, NY, USA, 210–215. https://doi.org/10.1145/3287624.3287706

[8] FastML Team. 2023. *fastmachinelearning/hls4ml*. https://doi.org/10.5281/zenodo.1201549

[9] Y. Serhan Gener, Sezer Gören, and H. Fatih Ugurdag. 2019. Lossless Look-Up Table Compression for Hardware Implementation of Transcendental Functions. In *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. 52–57. https://doi.org/10.1109/VLSI-SoC.2019.8920330

[10] Shen-Fu Hsiao, Kun-Chih Chen, and Yi-Hau Chen. 2018. Optimization of Lookup Table Size in Table-Bound Design of Function Computation. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4. https://doi.org/10.1109/ISCAS.2018.8350933

[11] Shen-Fu Hsiao, Chia-Sheng Wen, Yi-Hau Chen, and Kuei-Chun Huang. 2017. Hierarchical Multipartite Function Evaluation. *IEEE Trans. Comput.* 66, 1 (2017), 89–99. https://doi.org/10.1109/TC.2016.2574314

[12] Shen-Fu Hsiao, Po-Han Wu, Chia-Sheng Wen, and Pramod Kumar Meher. 2015. Table Size Reduction Methods for Faithfully Rounded Lookup-Table-Based Multiplierless Function Evaluation. *IEEE Transactions on Circuits and Systems II: Express Briefs* 62, 5 (2015), 466–470. https://doi.org/10.1109/TCSII.2014.2386232

[13] Alireza Khataei, Gaurav Singh, and Kia Bazargan. 2023. 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* (Monterey, CA, USA) *(FPGA '23)*. Association for Computing Machinery, New York, NY, USA, 165–175. https://doi.org/10.1145/3543622.3573181

[14] Alireza Khataei, Gaurav Singh, and Kia Bazargan. 2023. Optimizing Hybrid Binary-Unary Hardware Accelerators Using Self-Similarity Measures. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 105–113. https://doi.org/10.1109/FCCM57271.2023.00020

[15] Martin Langhammer and Bogdan Pasca. 2016. Single Precision Natural Logarithm Architecture for Hard Floating-Point and DSP-Enabled FPGAs. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. 164–171. https://doi.org/10.1109/ARITH.2016.20

[16] Jean-Michel Muller. 2020. Elementary Functions and Approximate Computing. *Proc. IEEE* 108, 12 (2020), 2136–2149. https://doi.org/10.1109/JPROC.2020.2991885

[17] Pragnesh Patel, Aman Arora, Earl Swartzlander, and Lizy John. 2022. LogGen: A Parameterized Generator for Designing Floating-Point Logarithm Units for Deep Learning. In *2022 23rd International Symposium on Quality Electronic Design (ISQED)*. 1–7. https://doi.org/10.1109/ISQED54688.2022.9806139

[18] M.J. Schulte and J.E. Stine. 1999. Approximating elementary functions with symmetric bipartite tables. *IEEE Trans. Comput.* 48, 8 (1999), 842–847. https://doi.org/10.1109/12.795125

[19] Yusheng Xie, Alex Noel Joseph Raj, Zhendong Hu, Shaohaohan Huang, Zhun Fan, and Miroslav Joler. 2020. A Twofold Lookup Table Architecture for Efficient Approximation of Activation Functions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 12 (2020), 2540–2550. https://doi.org/10.1109/TVLSI.2020.3015391