Efficient Table-based Function Approximation on FPGAs using Interval Splitting and BRAM Instantiation

CHETANA PRADHAN, MARTIN LETRAS, and JÜRGEN TEICH, Hardware/Software Co-Design, Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

This paper proposes a novel approach for the generation of memory-efficient table-based function approximation circuits for edge devices in general, and FPGAs in particular. Given a function $f(x)$ to be approximated in a given interval $[x_0, x_0 + a)$ and a maximum approximation error E_a, the goal is to determine a function table implementation with a minimized memory footprint, i.e., number of entries that need to be stored. Rather than state-of-the-art work performing an equidistant sampling of the given interval by so-called breakpoints and using linear interpolation between two adjacent breakpoints to determine $f(x)$ at the maximum error bound, we propose and compare three algorithms for splitting the given interval into sub-intervals to reduce the required memory footprint drastically based on the observation that in sub-intervals of low gradient, a coarser sampling grid may be assumed while guaranteeing the maximum interpolation error bound E_a. Experiments on elementary mathematical functions show that a large fraction in memory footprint may be saved. Second, a hardware architecture implementing the sub-interval selection, breakpoint lookup and interpolation at a latency of just 9 clock cycles is introduced. Third, for each generated circuit design, BRAMs are automatically instantiated rather than synthesizing the reduced footprint function table using LUT primitives providing an additional degree of resource efficiency. The approach presented here for FPGAs can equally be applied to other circuit technologies for fast and, at the same time, memory-optimized function approximation at the edge.

CCS Concepts: • **Hardware** → **Power and energy**; • **Computer systems organization** → **Embedded hardware**.

Additional Key Words and Phrases: FPGA, Approximate Computing, Function Approximation, BRAM

ACM Reference Format:
Chetana Pradhan, Martin Letras, and Jürgen Teich. 2022. Efficient Table-based Function Approximation on FPGAs using Interval Splitting and BRAM Instantiation. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (April 2022), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Approximate computing [1] is a new research field that investigates the trade-off between accuracy, latency, energy [1, 2], and cost of computations. Fig. 1 presents a comparison between approximate computing and conventional computing. Here, approximate computing primes high-performance at the expense of low accuracy. For example, many applications like video and image processing tolerate a certain degree of errors made during acquisition, processing and rendering of images.

Authors' address: Chetana Pradhan, chetana.pradhan@fau.de; Martin Letras, martin.letras@fau.de; Jürgen Teich, juergen.teich@fau.de, Hardware/Software Co-Design, Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany, Cauerstraße 11, Erlangen, Germany, 91058.

1539-9087/2022/4-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

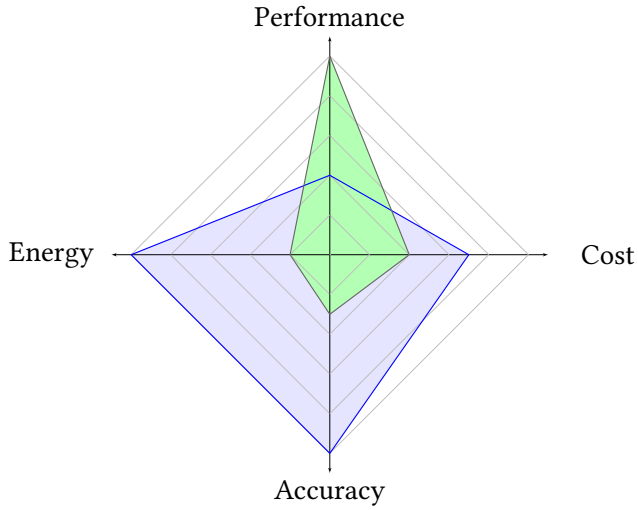


Fig. 1. Approximate computing (green) exploits the error tolerance and robustness of certain applications in order to trade-off accuracy against latency, energy and cost.

There already exists a plethora of work on approximate circuit design for basic arithmetic operations such as additions [3–5], multiplications [6, 7], or divisions [8]. However, much less effort has been invested on efficient implementation of function approximation in hardware including trigonometric, exponential, and logarithmic functions. Here, Taylor series expansions or iterative approximation techniques are well known and often applied, but these also come at either very high resource or latency demands. Tabular representations of functions can serve as an alternative solution in case of small quantization and approximation errors. Indeed, they play an important role in function approximation due to providing constant-time evaluations at the cost of high memory demands. E.g., Matlab/Simulink [9] already offers an available optimization framework named Look-up-table (LUT) Optimizer [10] that computes a function table approximation for a given mathematical function to be approximated within a given interval subject to a maximal approximation error. Notably, this framework also allows to semi-automatically generate code (C++ and VHDL) for tabular function approximations. Unfortunately, when synthesizing this VHDL code to a circuit implementation, e.g., for a Field Programmable Gate Array (FPGA) target, the tables are implemented quite inefficiently by using LUTs structures. Moreover, as [10] applies an equidistant sampling of a given interval, the table sizes can be prohibitively big to fit on an FPGA, particularly in case of low required approximation errors. Consequently, techniques for lowering the number of samples and thus table sizes are required. In addition, novel techniques for Hardware Description Language (HDL) generation and synthesis are needed for resource-efficient function approximation on modern FPGAs. One contribution of this paper is to instantiate internal so-called Block RAM (BRAM) structures [11]. These BRAMs can even be configured individually in terms of number of entries and bit width of each entry. For example, a so-called BRAM18 block can be alternatively configured to store 16,384 entries of 1 bit, 8,192 entries of 2 bits, or up to just 512 entries of 32 bits. Yet the main contribution of this paper is to introduce techniques for splitting a given domain of a function into distinct intervals to reduce the memory footprint of a function table that needs to be stored. This is accomplished by a technique called *interval splitting*. Three efficient interval splitting algorithms are introduced and compared based on the observation

that in sub-intervals of low gradient, a coarser sampling can be chosen to guard a given maximal approximation error.

In summary, this paper presents a novel table-based approach for function approximation proposing interval splitting to drastically reduce the memory footprint compared to a state-of-the-art method [10] but without any sacrifice in approximation error bounds. The contributions summarized in detail are:

- Introduction of three *interval splitting algorithms* based on the observation that in sub-intervals of low gradient, a coarser sampling grid may be assumed to satisfy a user-given maximum interpolation error bound E_a at any point x within a given interval $[x_0, x_0 + a)$ of interest. Accordingly, each sub-interval is assigned an individually optimized spacing between breakpoints. The overall memory footprint is minimized by assigning a coarser breakpoint spacing to sub-intervals with small slope. Only sub-intervals with larger slopes require a fine quantization in order to satisfy E_a. The proposed algorithms deliver a partition of the given interval into proper sub-intervals such that a maximum approximation error bound E_a is never violated over the whole interval range. It is shown that memory footprint reductions of up to 70 % in average are achieved over tables optimized and generated using the state-of-the-art tool LUT Optimizer by Matlab/Simulink [9].
- An *automated design flow* that uses the interval-based tabular function approximation to generate a hardware description in VHDL automatically. The proposed hardware circuit consists of three units. First, an interval selection circuit determines the index of the sub-interval containing the two breakpoints closest to x. Second, a table lookup unit that retrieves the two range values (y) of the breakpoints enclosing x. Finally, a linear interpolation is performed on these two looked-up values to return the approximation of $f(x)$. The whole architecture (depicted in Fig. 7) performs a function evaluation at a latency of 9 clock cycles.
- Moreover, instead of synthesizing the reduced footprint tables using LUTs, BRAMs are exploited and instantiated, providing an additional degree of resource efficiency.
- Finally, as a proof of concept, we present experimental results on the approximation of nine benchmark functions as test cases, three of them belonging to well-known activation functions for Artificial Neural Networks (ANNs) including the functions Sigmoid Linear Unit (Swish), Gaussian Error Linear Unit (GELU) and Softplus, also known as smooth Rectified Linear Unit (ReLU) function. The proposed methodology is shown to be able to synthesize constant low latency (9 clock cycles) circuit implementations for each test function. Particularly for ANN activation functions, resource-efficient circuit implementations are crucial due to the huge number of calculations of these functions during the training and inference phase of a neural network [12, 13].

The paper is structured as follows: Sec. 2 presents fundamentals and definitions. Sec. 3 then presents a reference approach according to [10] that will be used for comparison. Subsequently, Sec. 4 introduces the concept of interval splitting and presents three interval splitting algorithms. Sec. 5 presents the design flow and the proposed hardware architecture. Then, Sec. 6 presents the experimental results. Sec. 7 gives an overview on related work. Finally, Sec. 8 concludes our work.

2 FUNDAMENTALS

Storing the range values of a given function $f(x)$ for a discretized domain can help to avoid a computationally expensive polynomial approximation of $f(x)$. Unfortunately, this approach is impractical in case the given approximation error demands fine quantization of the domain. Indeed, the table size required to store the range values grows exponentially with the bit-width of x. E.g., if the bit-width for quantization of x is 8 bits, the size of the lookup table is $2^8 = 255$. However, if

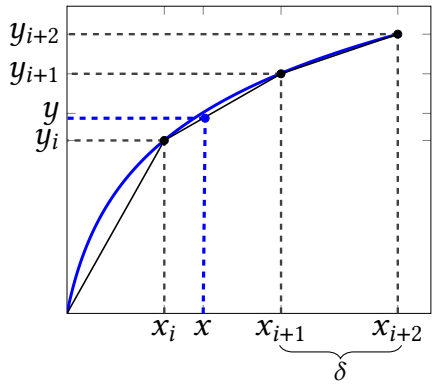


Fig. 2. Piecewise linear interpolation in between two breakpoints x_i and x_{i+1}.

the bit-width of x is 32, the lookup table contains $2^{32} = 4,294,967,296$ entries, thus resulting in a memory footprint of 16 GB. Consequently, this method cannot be applied for resource-constrained devices such as FPGAs due to the limited amount of available memory and other hardware resources. As a solution, the combination of tabular function representation and polynomial interpolation has emerged as a widespread technique for function approximation on FPGAs [14]. Instead of storing all the range values of $f(x)$ for a given quantization of x, a more efficient approach is to use a coarser quantization of x and perform polynomial approximation in between two adjacent points called breakpoints in the following. In [14], it is proposed to use a degree n polynomial $p(x)$ as follows:

$$p(x) = a_0 + a_1 x + \ldots + a_n x^n \tag{1}$$

Here, $a_i : 0 \leq i \leq n$ denotes the coefficients of the approximating polynomial $p(x)$. Accordingly, the number of entries of the table can be reduced. Still, this requires to store the $n + 1$ additional coefficients of each polynomial in between two adjacent breakpoints.
In the rest of this paper, we consider piecewise linear interpolation to avoid the increased computational complexities and memory requirements of higher degree polynomials. In general, breakpoints do not have to be located equidistantly. In this case, linear interpolation can be performed using Eq. (2). For example, the set of $k + 1$ breakpoints is denoted as $X = \{x_0, x_1, \ldots, x_k\}$. Y represents the range value of $f(x)$ for X as domain $Y = \{f(x_0), f(x_1), \ldots, f(x_k)\} = \{y_0, y_1, \ldots, y_k\}$. The first and last breakpoints $\{x_0, x_k = x_0 + a\} \in X$ enclose the interval of approximation. Fig. 2 illustrates the calculation of $f(x)$ for any given $x \in \mathbf{R}$. If x matches with any value in X, the result can be directly retrieved from a table storing the values of Y. Otherwise, the first step is to find the closest breakpoints x_i and x_{i+1} of the input. Their corresponding range values y_i and y_{i+1} are then looked up. Finally, the value of $f(x)$ is computed according to Eq. (2).

$$y = y_i + \frac{x - x_i}{x_{i+1} - x_i}(y_{i+1} - y_i) \tag{2}$$

If $x_{i+1} - x_i$ is not constant $\forall i \in \{0, 1, \ldots, k\}$, then the enclosing breakpoints can only be located by a search method. To this end, the number of comparisons performed grows with the number of breakpoints k. This can be avoided by even sampling of the interval $[x_0, x_0 + a)$. Now, when defining a uniform spacing $\delta = (x_{i+1} - x_i) > 0$ between two breakpoints, it is possible to determine

the index i describing the interval $[x_i, x_{i+1})$ that includes x as follows:

$$i = \left\lfloor \frac{(x - x_0)}{\delta} \right\rfloor \tag{3}$$

Here, only the first value x_0 and the spacing δ are required to calculate i. Finally, with $x_i = x_0 + i \cdot \delta$, Eq. (2) can be re-written to only use the points in Y, x_0, the location i according to Eq. (3) and the spacing δ:

$$y = y_i + \frac{x - (x_0 + i \cdot \delta)}{\delta}(y_{i+1} - y_i) \tag{4}$$

Thus, the function approximation can be achieved just by storing the $k + 1$ range values of X. Henceforth, we only consider equidistant spacing between two breakpoints. The required memory footprint is referred to as M_F in the following sections. For the above linear interpolation scheme, we obtain a memory footprint M_F of:

$$M_F = k + 1 \tag{5}$$

3 REFERENCE APPROACH

In this section, we introduce the *Reference* approach [10] used later (see Sec. 6) for comparison. According to Eq. (5), the memory footprint M_F is linear in k, the number of breakpoints. An obvious remaining problem here is how to determine an equidistant spacing δ being as large as possible such that still a user-given maximal approximation error bound E_a is never exceeded for any evaluation of $f(x)$ inside the given interval $[x_0, x_0 + a)$. Let $p_i(x)$ denote the linear polynomial used to approximate the function $f(x)$ between the adjacent breakpoints $[x_i, x_{i+1})$. A two-point line expression can be derived from Eq. (2) as follows:

$$p_i(x) = p_i(x_i) + \frac{x - x_i}{x_{i+1} - x_i}(p_i(x_{i+1}) - p_i(x_i)) \tag{6}$$

If the second derivative $f''(x)$ of $f(x)$ does exist at each point in $[x_i, x_{i+1})$, then the difference between the exact function value $f(x)$ and the value of the approximating polynomial $p_i(x)$ in $x_i \leq x < x_{i+1}$ is given as:

$$f(x) - p_i(x) = \frac{(x - x_i)(x - x_{i+1})}{2} f''(\varepsilon_x) \tag{7}$$

In Eq. (7), ε_x is some value between x_i and x_{i+1}. In consequence, an error bound can be formulated based on Eqs. (6) and (7) as:

$$|f(x) - p_i(x)| \leq \frac{(x - x_i)(x_{i+1} - x)}{2} \max_{x_i \leq x < x_{i+1}} |f''(x)| \tag{8}$$

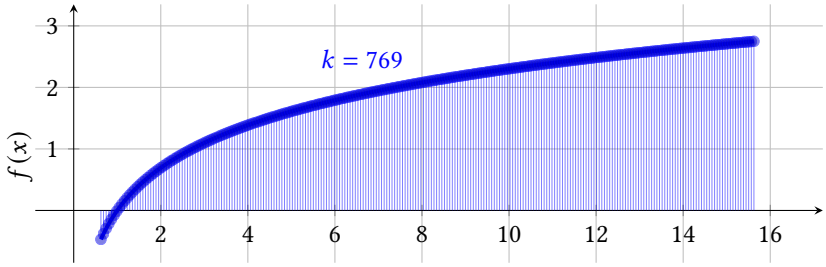
With the spacing $\delta_i = x_{i+1} - x_i$ between x_i and x_{i+1}, the maximum value of $(x - x_i)(x_{i+1} - x)$ in Eq. (8) can be constrained as:

$$\max_{x_i \leq x < x_{i+1}} (x - x_i)(x - x_{i+1}) = \frac{\delta_i^2}{4} \tag{9}$$

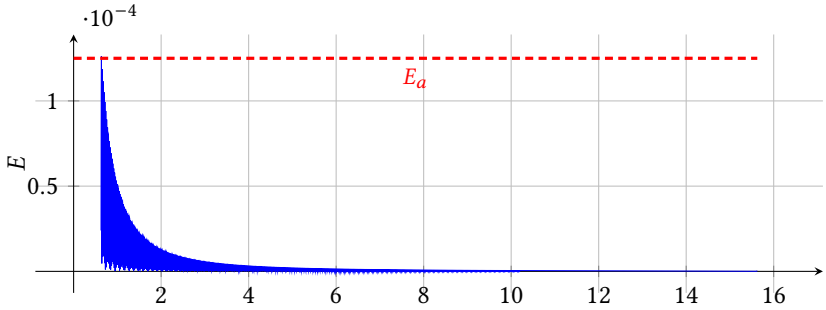
By combining Eqs. (8) and (9), we obtain a maximum approximation error bound E_i given a spacing δ_i:

$$E_i = \frac{\delta_i^2}{8} \max_{x_i \leq x < x_{i+1}} |f''(x)| \tag{10}$$

The dependence of the approximation error on the second derivative of a function is intuitive from the perspective of linearity. If f is truly linear in the interval $[x_i, x_{i+1})$, then the second derivative vanishes implying an exact representation. The value of $\max_{x_i \leq x < x_{i+1}} |f''(x)|$ can be expressed in closed



(a) Table-based approximation of $f(x) = log(x)$. Here, $\delta = 0.019$, resulting in a memory footprint of $M_F = k + 1 = 770$ entries.



(b) Approximation Error obtained using Eq. (11) given $E_a = 1.25E - 04$.

Fig. 3. Function approximation of $f(x) = log(x)$ in the interval $[0.625, 15.625)$.

form for elementary functions as well as some of the non-elementary functions due to their well-defined second derivatives. Finally, for a given user-defined maximal approximation error bound E_a to hold between any pair of breakpoints and assuming equi-distant spacings $\delta = \delta_i, 0 \leq i \leq k-1$, we can infer the biggest permissible spacing δ from the segment i with the smallest value of δ_i in Eq. (10):

$$\delta(f, E_a, [x_0, x_k)) = \min_{0 \leq i \leq k-1} \left(8 \cdot \frac{E_a}{\max\limits_{x_i \leq x < x_{i+1}} |f''(x)|} \right)^{\frac{1}{2}} \tag{11}$$

E.g., Fig. 3 illustrates the approximation of $f(x) = log(x)$ in the interval $[0.625, 15.625)$ and $E_a = 1.25E - 04$. This results in an spacing $\delta(log(x), E_a, [0.625, 16.625)) = 0.019$. From a spacing δ and the interval $[x_0, x_0 + a)$, it is possible to generate the entries to be stored in a lookup table (see Fig. 3a). Given a spacing δ and an interval $[x_0, x_0 + a)$, the memory footprint of this approach called *Reference* approach in the following can be calculated as:

$$M_F^R(\delta, [x_0, x_0 + a)) = \left\lceil \frac{x_0 + a - x_0}{\delta} \right\rceil + 1 \tag{12}$$

Thus, the memory footprint of $f(x) = log(x)$, given the interval $[0.625, 15.625)$, and $\delta = 0.019$ is $M_F^R(0.019, [0.626, 15.625)) = 770$ entries.

The *Reference* approach performs the function approximation given a maximum approximation error E_a by delivering a set of evenly spaced breakpoints. However, this approach does not take into account the gradient in different regions of the interval. This may result in an inefficient footprint, e.g., in Fig. 3, the interval closest to the left-most point $x_0 = 0.625$ determines the maximal spacing for the given approximation error E_a. However, it can be seen that for larger values of x (see Fig. 3b),

a coarser spacing could be used when splitting the domain into sub-intervals and using different spacings δ_i within these. Our main idea for the three proposed approaches introduced in the following is therefore to use different spacings δ_i by splitting the given interval into sub-intervals.

4 INTERVAL SPLITTING ALGORITHMS

Uniform spacing schemes such as even spacing (see the *Reference* approach in Sec. 3) or power of two spacing do not consider the local variation of many functions in the interval of approximation. This section introduces three algorithmic approaches to partition a given interval $[x_0, x_0 + a)$ into a set of non-uniform sub-intervals and determine a uniform spacing of breakpoints in each sub-interval given a maximum tolerable approximation error E_a. The proposed algorithms perform the segmentation of the interval $[x_0, x_0 + a)$ into a partition of sub-intervals exploiting the granularity in the spacing according to the steepness of a given function. Here, we trade between the number of generated sub-intervals and the memory footprint reduction. The presented algorithms differ mainly in the heuristic proposed to compute the set of sub-interval partitions. The first two algorithms (Algorithms 1 and 2) split a given interval $[x_0, x_0 + a)$ into two sub-intervals recursively until a corresponding stopping criterion is satisfied. Algorithm 1, called *binary segmentation* always splits a sub-interval at the midpoint. In contrast, Algorithm 2, called *hierarchical segmentation* finds the best splitting point by sweeping over the given interval such that the reduction in memory footprint M_F when splitting is maximized. Here, the step size for the sweep is given as an input. Finally, Algorithm 3 named *sequential segmentation* also performs a sweep-based interval splitting. However here, interval splitting is performed in a single sweep over the overall interval $[x_0, x_0 + a)$.

4.1 Binary Segmentation

Algorithm 1 performs a *binary segmentation* of a given interval $[p_i, p_{i+1})$ in which the function $f(x)$ is to be approximated. The inputs of Algorithm 1 are the function $f(x)$, the interval $[x_0, x_0 + a)$, the maximum approximation error E_a and a threshold value ω which determines whether a new sub-interval split is accepted. *Binary segmentation* determines a partition P of sub-intervals, from which a set of spacings S and a set K containing the numbers of breakpoints in each sub-interval can be obtained. Each element of $p_i \in P$ represents a left (sub-interval) delimiter value. The number of sub-intervals is thus $|P| - 1$. As an example for illustration, assume Algorithm 1 determines the sub-interval splitting $P = \{p_0, p_1, p_2\}$ for a given function, that yields the sets of values $S = \{\delta_0, \delta_1\}$ and $K = \{\kappa_0, \kappa_1\}$. P represents a partitioning of a given interval into two sub-intervals $[p_0, p_1)$ and $[p_1, p_2)$. Here, δ_0 and κ_0 correspond to the spacing and the number of breakpoints for the sub-interval $[p_0, p_1)$ as well as δ_1 and κ_1 correspond to sub-interval $[p_1, p_2)$. The memory footprint corresponding to P is therefore given by:

$$M_F^P([p_0, p_{|P|-1})) = \sum_{j=0}^{|P|-1} \kappa_j \tag{13}$$

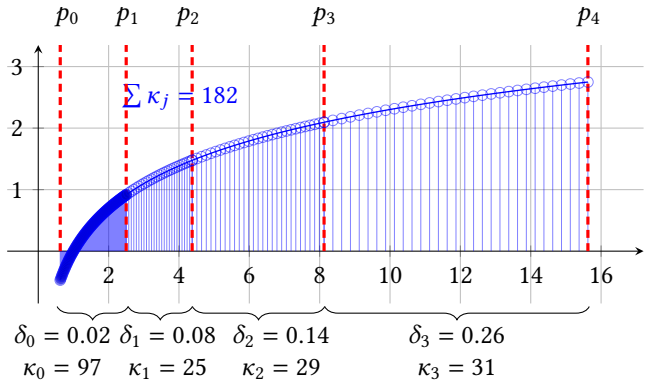
Binary segmentation is a recursive algorithm, which evaluates the reduction in M_F obtained by splitting the current interval $[p_i, p_{i+1})$ at the midpoint bp. Initially, the lower and upper bounds of the interval of approximation ($[x_0, x_0 + a)$) are provided as the inputs $p_i = x_0$ and $p_{i+1} = x_0 + a$ to the function *Binary*. The first step is to initialize P as $\{p_i, p_{i+1}\}$ (see Line 4 in Algorithm 1). The spacing δ_p and the number of breakpoints κ_p are obtained using the *Reference* approach in the interval $[p_i, p_{i+1})$ (see Sec. 3).

The midpoint bp of the input interval is used to create the left sub-interval $bp_1 = [p_i, bp)$ and right sub-interval $bp_2 = [bp, p_{i+1})$. The spacings δ_{bp_1} and δ_{bp_2}, and the number of breakpoints

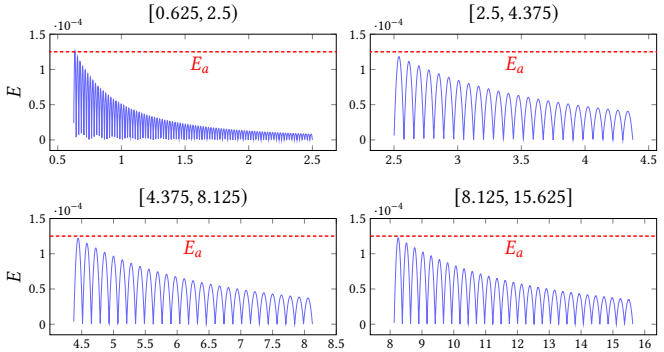
Algorithm 1: Binary Interval Splitting $[p_i, p_{i+1})$

1 **Function** `Binary`(ω,E_a,f,p_i,p_{i+1})
  Input: f is the function to be approximated
    p_i is the lower bound of the interval
    p_{i+1} is the upper bound of the interval
    ω is the reduction threshold (0,1]
    E_a is the maximum approximation error
  Output: $P = \{p_0, p_1, \ldots, p_n\}$ is the set of sub-interval boundaries
2 **begin**
3 $P \leftarrow \{p_i, p_{i+1}\}$ // Initial set of sub-intervals boundaries
4 $\delta_p \leftarrow \delta(f, E_a, [p_i, p_{i+1}))$ // Calculate the spacing in the interval
5 $\kappa_p \leftarrow M_F(\delta_p, [p_i, p_{i+1}))$ // Calculate the M_F of the interval
6 $bp \leftarrow \frac{p_i + p_{i+1}}{2}$ // Midpoint between p_i and p_{i+1}
7 $\delta_{bp_1} \leftarrow \delta(f, E_a, [p_i, bp))$ // Calculate the spacing of sub-intervals
8 $\delta_{bp_2} \leftarrow \delta(f, E_a, [bp, p_{i+1}))$
9 **if** $\delta_{bp_1} \neq \delta_{bp_2}$ **then**
10 $\kappa_{bp_1} \leftarrow M_F(\delta_{bp_1}, [p_i, bp))$ // Calculate the M_F of sub-intervals
11 $\kappa_{bp_2} \leftarrow M_F(\delta_{bp_2}, [bp, p_{i+1}))$
12 **if** $\kappa_{bp_1} + \kappa_{bp_2} < \kappa_p \cdot \omega$ **then** // If **true**, accept sub-interval split
13 $P \leftarrow \{\texttt{Binary}(\omega, E_a, f, p_i, bp) \cup \texttt{Binary}(\omega, E_a, f, bp, p_{i+1})\}$
        // Recursive call for sub-intervals $[p_i, bp)$ and $[bp, p_{i+1})$
14 **end**
15 **end**
16 **return** P
17 **end**
18 **End Function**

κ_{bp_1} and κ_{bp_2} of sub-intervals bp_1 and bp_2 are then calculated according to Eq. (11) and Eq. (12) (see Lines 8-12 in Algorithm 1). If the sum of κ_{bp_1} and κ_{bp_2} denoting the memory footprints of bp_1 and bp_2 is less than a specified fraction (threshold ω) of κ_p (see Line 13 in Algorithm 1), the memory footprint reduction and thus split are accepted and the *binary* function is called recursively for the sub-intervals bp_1 and bp_2, respectively. Otherwise, the algorithm terminates and returns the current input interval boundaries ($\{p_i, p_{i+1}\}$). The final set P returned is the union of all the returned sub-intervals. The value $\omega \in (0, 1]$ indicates the threshold of an acceptable memory footprint reduction. E.g., $\omega = 0.3$ indicates that an interval split must lead to a footprint reduction of at least 30 % of the given interval in order to continue splitting the left and right sub-intervals. Fig. 4 illustrates the proposed recursive sub-interval splitting approximation method using as an example, the function $f = log(x)$ to be approximated in the interval $[p_i, p_{i+1}) = [0.625, 15.625)$ with a maximum approximation error of $E_a = 1.22E-04$ and splitting threshold $\omega = 0.3$. To find a partition of the interval, *binary segmentation* is performed with the inputs ω, E_a, f and $[p_i, p_{i+1})$. Using the *Reference* approach, the uniform spacing δ_p and memory footprint M_F are obtained as $\delta_p = 0.019$ and $M_F = 770$, respectively. When applying Algorithm 1, the middle point $bp = 8.125$ of the interval $[0.625, 15.625)$ is determined first. The left and right sub-intervals are derived using bp, thus $bp_1 = [0.625, 8.125)$ and $bp_2 = [8.125, 15.625)$. After calculating the spacing $\delta_{bp_1} = 0.0195$ and $\delta_{bp_2} = 0.25$ for these two sub-intervals, respectively, the number of breakpoints results in $\kappa_{bp_1} = 384$ and $\kappa_{bp_2} = 31$. The new partition has a memory footprint of $M_F^P = (384 + 31) = 415$.



(a) Function approximation of $log(x)$ obtained by *binary segmentation*.



(b) Error margins obtained by *binary segmentation*.

Fig. 4. For a user-given maximal approximation error bound of $E_a = 1.22E-04$, the algorithm *binary segmentation* identifies the partition $P = \{0.625, 2.5, 4.375, 8.125, 15.625\}$ for splitting threshold $\omega = 0.3$ describing four sub-intervals $[0.625, 2.5)$, $[2.5, 4.375)$, $[4.375, 8.125)$ and $[8.125, 15.625)$. Here, a total of $M_F = 182$ entries represents an overall reduction in memory footprint of 76 % compared to the *Reference* approach (see Fig. 3a).

Compared to the memory footprint of 770, the achieved reduction is 46 % which is greater than the required threshold of 30 % ($\omega = 0.3$). Then, the function *binary* is called recursively for the accepted right and left sub-intervals $[0.625, 8.125)$ and $[8.125, 15.625)$, respectively. The previous is repeated until no more splittings can be achieved with an acceptable memory footprint reduction. Algorithm 1 stops with the final partition $P = \{0.625, 2.5, 4.375, 8.125, 15.625\}$. Hence, four sub-intervals $[0.0625, 2.5)$, $[2.5, 4.375)$, $[4.375, 8.125)$, and $[8.125, 15.625)$ were identified (see Fig. 4). The spacings and the number of breakpoints of each sub-intervals are $S = \{0.02, 0.08, 0.13, 0.25\}$ and $K = \{97, 25, 29, 31\}$. From the set of memory footprints K, the overall memory footprint $M_F^P([0.625, 15.625)) = (97 + 25 + 29 + 31) = 182$ is calculated. Compared to the reported memory footprint $M_F^R = 770$ for the same function, interval, and error bound (see Fig. 3) using the *Reference* approach, *binary segmentation* is thus able to reduce the memory footprint by 76 % while respecting the maximum approximation error bound imposed by the given E_a (see Fig. 4b).
However, using the midpoint in the partitioning heuristic might lead to a sub-optimal exploration of the partitions in approximation of a given function $f(x)$. The next section introduces an alternative

recursive approach which employs a different heuristic in attempt to achieve even larger memory footprint reductions.

4.2 Hierarchical Segmentation

Hierarchical segmentation is an alternative recursive interval splitting heuristic described by the function *Hierarchical* in Algorithm 2. For each candidate interval $[p_i, p_{i+1})$, a linear sweep is performed instead of just using the midpoint to split the interval. *Hierarchical segmentation* then chooses the splitting point sp with the highest memory footprint reduction and performs a split at sp, if the reduction is acceptable. In order to reduce the computational effort, a step size ε denotes the uniform distance between two adjacent splitting point candidates. The value of ε, the function f, the interval $[p_i, p_{i+1})$, the reduction threshold ω, and the maximum approximation error E_a are passed as inputs to Algorithm 2 (see Line 2 of Algorithm 2). The output is the set P, from which a set of spacings S and a set of number of breakpoints K are determined (see Line 3 of Algorithm 2). Similar to Algorithm 1, Algorithm 2 initializes the set $P = \{p_i, p_{i+1}\}$ with the lower and upper bounds of a given input interval $[p_i, p_{i+1})$ (see Line 4 in Algorithm 2). The algorithm then computes the maximal number of splitting point candidates j_{max} based on the step size ε as a parameter (see Line 5 in Algorithm 2). In the next step, we evaluate all the possible memory footprints by splitting the input interval at each candidate point. The splitting point sp is then chosen as the point delivering the smallest memory footprint (see Lines 6 and 7 in Algorithm 2). The left and right sub-intervals are derived as $sp_1 = [p_i, sp)$ and $sp_2 = [sp, p_{i+1})$, respectively. The spacings δ_{sp_1} and δ_{sp_2} and number of breakpoints κ_{sp_1} and κ_{sp_2} of sub-intervals sp_1 and sp_2 are calculated using the *Reference* approach (see Lines 9-12 in Algorithm 2). If the sum of κ_{sp_1} and κ_{sp_2} denoting the memory footprint of an interval split at the position sp is less than a specified fraction ω of κ_p (the memory footprint over the interval $[p_i, p_{i+1})$), the split is accepted and the hierarchical function is called recursively to explore the right and left sub-intervals (see Line 13 in Algorithm 2). Otherwise, the algorithm terminates returning the current upper and lower interval bounds (see Line 16 in Algorithm 2). The final set P is then returned as the union of split intervals.

Fig. 5a shows the hierarchical segmentation approach over the interval $[p_i, p_{i+1}) = [0.625, 15.625)$. E.g., let $f(x) = log(x)$, the reduction threshold $\omega = 0.3$, the maximum absolute error $E_a = 1.22E{-}04$, and the sweep size $\varepsilon = 0.015$. At this point, the number of candidate splitting points is $j_{max} = \frac{15.625-0.625}{0.015} = 1000$. This set of points can be expressed in terms of ε and j_{max} as $\{p_i + j \cdot \varepsilon | i \leq j < j_{max}\}$. After evaluating each candidate, Algorithm 2 determines $sp = 2.90$ as the best candidate resulting in the left sub-interval $sp_1 = [0.625, 2.90)$ and the right sub-interval $sp_2 = [2.90, 15.625)$. κ_{sp_1} is equal to 117 and κ_{sp_2} is 141. The partition has a memory footprint of $\kappa_{sp_1} + \kappa_{sp_2} = (117 + 141) = 258$ which implies a 66.5 % reduction in memory footprint compared to M_F^R obtained by the *Reference* approach. The achieved reduction is greater than the required threshold reduction of 30 % ($\omega = 0.3$). Thus, the recursive splitting continues for the sub-intervals sp_1 and sp_2. The previous steps are repeated until no more splitting can be performed resulting in the set $P = \{0.6250, 1.2106, 2.9073, 6.2556, 15.6250\}$, as illustrated in Fig. 5a where the set of spacings $S = \{0.01, 0.06, 0.09, 0.19\}$, and the set of breakpoints $K = \{30, 25, 37, 49\}$. The value of $M_F^P([0.625, 15.625)) = 30 + 45 + 37 + 49 = 161$. Hierarchical segmentation is able to reduce the memory footprint by 79 % with respect to *Reference* approach and by 11.5 % compared to binary segmentation (Algorithm 1).

4.3 Sequential Segmentation

Algorithm 3 describes the third approach called *sequential segmentation* and receives as inputs also the function $f(x)$ to approximate, the interval $[x_0, x_0 + a)$, maximum approximation error E_a, the reduction threshold ω and a sweep step size ε. Sequential segmentation performs a linear sweep of

Algorithm 2: Hierarchical Interval Splitting $[p_i, p_{i+1})$

1 **Function** `Hierarchical`(ω,E_a,f,p_i,p_{i+1})
  Input: f is the function to be approximated
    p_i is the lower bound of the interval
    p_{i+1} is the upper bound of the interval
    ω is the reduction threshold (0,1]
    E_a is the maximum approximation error
    ε is the sweep step size
  Output: $P = \{p_0, p_1, \ldots, p_n\}$ is set of sub-interval boundaries
2 **begin**
3 $P \leftarrow \{p_i, p_{i+1}\}$ `// Initial set of sub-intervals boundaries`
4 $j_{max} \leftarrow \left\lfloor \frac{p_{i+1} - p_i}{\varepsilon} \right\rfloor$ `// Calculate the number of splitting point candidates`
5 $j^* \leftarrow \underset{1 \le j \le j_{max}}{\arg\min} M_F(\delta(f, E_a, [p_i, p_i + j \times \varepsilon)), [p_i, p_i + j \times \varepsilon)) + M_F(\delta(f, E_a, [p_i + j \times \varepsilon, p_{i+1})), [p_i + j \times \varepsilon, p_{i+1}))$
6 $sp \leftarrow p_i + j^* \times \varepsilon$ `// Determine the splitting point`
7 $\kappa_p \leftarrow M_F(\delta_p, [p_i, p_{i+1}))$ `// Calculate the` M_F `of the interval`
8 $\delta_{sp_1} \leftarrow \delta(f, E_a, [p_i, sp))$ `// Calculate the spacing of sub-intervals`
9 $\delta_{sp_2} \leftarrow \delta(f, E_a, [sp, p_{i+1}))$
10 $\kappa_{sp_1} \leftarrow M_F(\delta_{sp_1}, [p_i, sp))$ `// Calculate the` M_F `of sub-intervals`
11 $\kappa_{sp_2} \leftarrow M_F(\delta_{sp_2}, [sp, p_{i+1}))$
12 **if** $\kappa_{sp_1} + \kappa_{sp_2} < \kappa_p \cdot \omega$ **then** `// If` **true**`, accept interval split`
13 $P \leftarrow \{$`Hierarchical`$(\omega, E_a, f, p_i, sp, \varepsilon) \cup$ `Hierarchical`$(\omega, E_a, f, sp, p_{i+1}, \varepsilon)\}$
      `// Recursive call for sub-intervals` $[p_i, sp)$ `and` $[sp, p_{i+1})$
14 **end**
15 **return** P
16 **end**
17 **End Function**

the overall input interval similar to the hierarchical segmentation. However, it produces a partition P of a given interval $[x_0, x_0 + a)$ iteratively in a single sweep. The set of splitting point candidates can be represented as $\{sp \mid sp = x_0 + i \cdot \varepsilon\}$ where $1 \le i < i_{max}$. Here, the distance between two adjacent points in the set is ε and the number of candidates i_{max} is determined based on the length of interval a and the step size ε (see Line 2 in Algorithm 3).

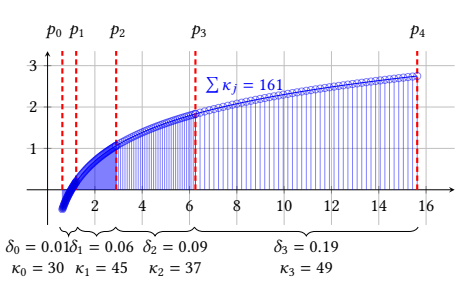
Algorithm 3 initializes P and x_p with the lower bound x_0 of the input interval (see Lines 1-2 in Algorithm 3). The value of x_p always corresponds to the last entry of the set of partitions P. The spacing δ_p and the memory footprint κ_p of the input interval $[x_0, x_0 + a)$ are calculated following the *Reference* approach (Sec. 3). The algorithm iterates through the sweep candidates sp and calculates the memory footprints κ_{sp_1} and κ_{sp_2} of the sub-intervals $sp_1 = [x_p, sp)$ and $sp_2 = [sp, x_0 + a)$, respectively. If the memory footprint of the input interval $[x_0, x_0 + a)$ split at sp is less than a fraction ω of κ_p, then the set P is updated with sp as a new splitting point. The values x_p, δ_p and κ_p are updated accordingly (see Lines 13-16 in Algorithm 3). Algorithm 3 stops with the last sweep value which is one step size smaller than $x_0 + a$.

The main difference between *sequential segmentation* and *binary* as well as *hierarchical segmentation* is how the partition points are generated. Here, *sequential segmentation* sweeps from left to right to find the set of partitions, and only one iteration over the given interval is required. In contrast, *binary* and *hierarchical* perform a recursive exploration of each split interval. Every time a new partition point is found, two new sub-intervals located at the right and the left are generated and

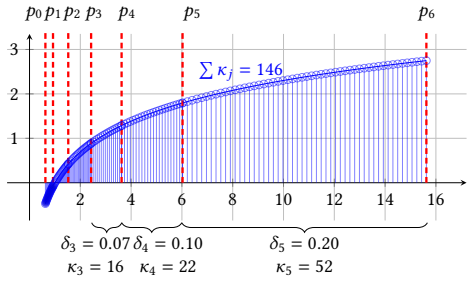
Algorithm 3: Sequential Interval Splitting $[x_0, x_0 + a)$

Function `Sequential`(ω,E_a,f,x_0,$x_0 + a$)
  Input: f is the function to be approximated
    x_0 is the lower bound of the interval
    $x_0 + a$ is the upper bound of the interval
    ω is the reduction threshold (0,1]
    E_a is the maximum approximation error
    ε is the sweep step size
  Output: $P = \{p_0, p_1, \ldots, p_n\}$ is the set of sub-interval boundaries
  $P \leftarrow \{x_0\}$ // Initialize P with the lower bound of the interval
  $x_p \leftarrow x_0$
  $i_{max} \leftarrow \left\lfloor \frac{a}{\varepsilon} \right\rfloor$ // Calculate the number of splitting point candidates
  $\delta_p \leftarrow \delta(f, E_a, [x_p, x_0 + a))$ // Calculate the spacing of sub-interval $[x_p, x_0 + a]$
  $\kappa_p \leftarrow M_F(\delta_p, [x_p, x_0 + a))$ // Calculate the M_F of sub-interval $[x_p, x_0 + a]$
  for $i \leftarrow 1$ **to** i_{max} **do**
    $sp \leftarrow x_0 + i \cdot \varepsilon$ // Determine the splitting point
    $\delta_{sp_1} \leftarrow \delta(f, E_a, [x_p, sp))$ // Calculate the spacing of sub-intervals
    $\delta_{sp_2} \leftarrow \delta(f, E_a, [sp, x_0 + a))$
    $\kappa_{sp_1} \leftarrow M_F(\delta_{sp_1}, [x_p, sp))$ // Calculate the M_F of sub-intervals
    $\kappa_{sp_2} \leftarrow M_F(\delta_{sp_2}, [sp, x_0 + a))$
    if $\kappa_{sp_1} + \kappa_{sp_2} < \kappa_p \cdot \omega$ **then** // If **true**, accept the interval split
      $P \leftarrow P \cup \{sp\}$ // Updating P
      $x_p \leftarrow sp$ // Updating x_p
      $\delta_p \leftarrow \delta(f, E_a, [x_p, x_0 + a))$ // Calc. the spacing
      $\kappa_p \leftarrow M_F(\delta_p, [x_p, x_0 + a))$ // Calc. the M_F of the interval to split
    end
  end
  $P \leftarrow P \cup \{x_0 + a\}$ // Updating P with the upper bound of the interval
End Function

explored until no more acceptable memory footprint reductions are achieved.
Fig. 5b presents the partition obtained by the algorithm *sequential segmentation* for $f(x) = log(x)$, over the interval $[0.625, 15.625)$, the reduction threshold $\omega = 0.3$, maximum absolute error $E_a = 1.22E-04$, and the sweep size $\varepsilon = 0.3$. Algorithm 3 determines the first splitting point as $sp = 0.9250$ while sweeping the interval $[0.625, 15.625)$. Thus, resulting in the new sub-intervals $sp_1 = [0.625, 0.925)$ and $sp_2 = [0.925, 15.625)$ with a memory footprint of $\kappa_{sp_1} = 16$ and $\kappa_{sp_2} = 510$, respectively. The partition has a memory footprint of $\kappa_{sp_1} + \kappa_{sp_2} = (16 + 510) = 526$ which results in a reduction of 31.6 % compared to the *Reference* approach and is therefore accepted because the memory footprint reduction obtained is greater than the required threshold of 30 % ($\omega = 0.3$). The previous steps are repeated for all the sweep values sp in the interval $[0.625, 15.625)$. Fig. 5b shows that the completed sweep over the interval $[0.625, 15.625)$ produces the partition $P = \{0.625, 0.925, 1.525, 2.425, 3.625, 6.025, 15.625\}$, resulting in a memory footprint $M_F^P([0.625, 15.625)) = (16 + 21 + 19 + 16 + 22 + 52) = 146$. *Sequential segmentation* is able to reduce the memory footprint by 81 % with respect to the *Reference* approach. Compared to the *binary* and *hierarchical* segmentation approaches, the memory footprint reduction is higher by 18 % and 9 %, respectively. Finally, *sequential segmentation* generates six sub-intervals which is larger than the



(a) Function approximation of $log(x)$ obtained using *hierarchical segmentation*.



(b) Function approximation of $log(x)$ obtained using *sequential segmentation*.

Fig. 5. Algorithm *hierarchical segmentation* identifies the partition $P = \{0.625, 1.21, 2.90, 6.25, 15.625\}$ for a given splitting threshold of $\omega = 0.3$ describing four sub-intervals $[0.625, 1.21)$, $[1.21, 2.90)$, $[2.90, 6.25)$ and $[6.25, 15.625)$. Here, a total of $M_F = 161$ entries represents an overall reduction in memory footprint of 79 % compared to the *Reference* approach (see Fig. 3a) for a user-given maximal approximation error bound of $E_a = 1.22E - 04$. On the other hand, the *sequential segmentation* obtained the partition $P = \{0.625, 1.21, 2.90, 6.25, 15.625\}$ with $M_F = 146$ resulting in a memory footprint reduction of 81 % compared to the *Reference* approach.

number of sub-intervals produced by the other two heuristics.
The following section presents an analysis of the proposed approaches to determine if an algorithm delivers significantly better memory footprint reductions than another one.

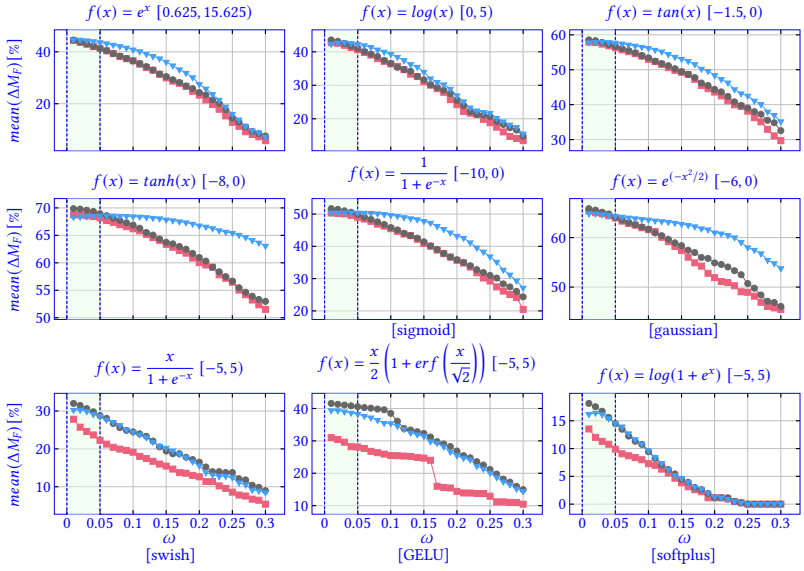
4.4 Comparative Analysis of the Proposed Approaches

As shown by the previous three examples (see Figs. 4 and 5), the proposed partitioning heuristics can achieve significant overall memory footprint reductions by splitting the interval of approximation. This section presents the comparison of the three presented approaches in terms of the memory footprint reductions compared to the *Reference* approach. We define the memory footprint reduction as:

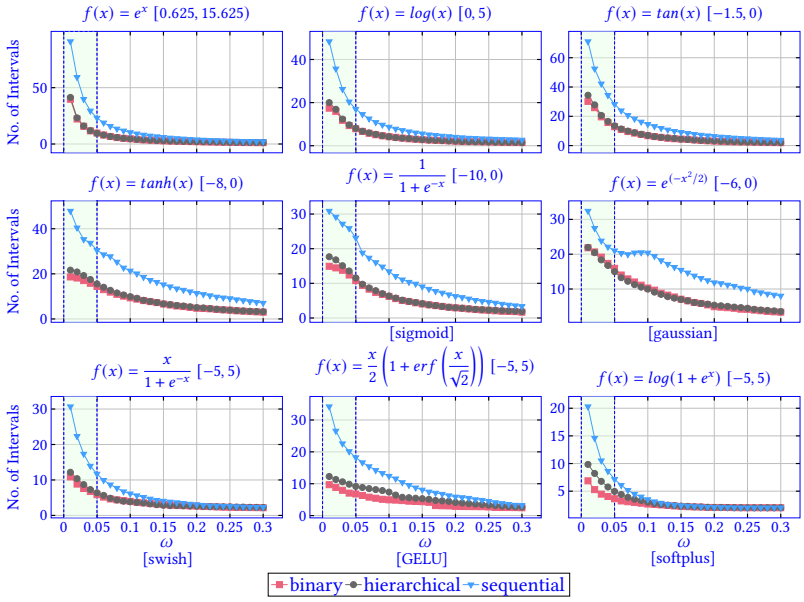
$$\Delta M_F[\%] = \frac{M_F^R - M_F^P}{M_F^R} \times 100 \tag{14}$$

In Eq. (14), M_F^R and M_F^P correspond to the memory footprints obtained respectively by the *Reference* approach and any of the three proposed approaches.

In Fig. 6, we compare the effects of varying the reduction threshold ω on the memory footprint reduction ΔM_F and on the number of intervals generated by each of our proposed approaches. Here, nine functions including several elementary mathematical functions, but also multiple well-known activation functions for ANNs, are used as test cases (see also Table 1). Next to each function, the interval $[x_0, x_0 + a)$ used for approximation is specified. Fig. 6a shows the memory footprint evaluation of the binary, hierarchical, and sequential segmentation colored in red, gray, and blue for each of the nine considered test functions. The *x-axis* corresponds to the reduction threshold ω varying from ω=0.01 to ω=0.3 (1 % to 30 % of memory footprint reductions). The *y-axis* represents the mean memory footprint reduction over a population X of 100 randomly generated intervals contained in the interval $[x_0, x_0 + a)$. All the nine test case functions are approximated while guarding a maximum absolute error $E_a = 9.5367E - 07$. Fig. 6b presents the number of generated sub-intervals for each of the three proposed approaches for each evaluated value ω for the same test functions as shown in Fig. 6a.



(a) Memory footprint reduction analysis of the proposed binary, hierarchical and sequential approaches.



(b) Number of intervals obtained by the proposed binary, hierarchical and sequential approaches.

Fig. 6. Memory footprint reduction analysis of the proposed binary, hierarchical and sequential interval-based segmentation approaches colored in red, gray, and blue, respectively. The *x-axis* presents 30 threshold values ω ranging from 0.01 to 0.3. Each point represents the mean memory footprint reduction $mean(\Delta M_F)$ in (a) and the mean number of sub-intervals determined in (b) for 100 randomly generated sub-intervals in the interval range $[x_0, x_0 + a)$ for each given value ω.

Fig. 6a analyzes the impact of the reduction threshold ω on $mean(\Delta M_F)$ obtained by the proposed approaches. As can be seen, all three approaches produce the highest memory footprint reductions and the highest number of generated intervals (see the left part of each plot in Fig. 6a and Fig. 6b) for the smallest values of ω. Accordingly, a fine-grained exploration of the partitions over the interval is performed. For higher values of the threshold parameter ω, the number of generated intervals reduces but going hand in hand with a lower reduction in the memory footprint (see the right part of each plot in Fig. 6a and Fig. 6b).

Fig. 6 lets us also identify the threshold regions of highest memory footprint reduction as well as the best interval splitting algorithm. Independent of the test function and interval splitting approach used, the highest absolute memory footprint reductions are consistently obtained for the smallest values of the threshold ω (see green shaded area $0.01 \leq \omega \leq 0.05$ in Fig. 6a). Now, as the memory footprint reduction of the resulting function table is the primary optimization goal for minimizing the table size[1], we can conclude that the hiearchical segmentation approach is the preferred choice for giving the highest memory footprint reductions at a lowest number of intervals for all nine test functions and that low ω values in the range $0.01 \leq \omega \leq 0.05$ will deliver the highest memory footpring reductions.

5 HARDWARE IMPLEMENTATION

Our second major contribution is the introduction of a generic, automatically synthesizable hardware implementation of the proposed approaches for the table-based function approximation. Fig. 7 depicts this hardware architecture.

The architecture's input is a bit vector Ξ to be evaluated by the approximation of $f(x)$. The architecture's output is a bit vector Υ. The input and the output are assumed to be user-defined as fixed point numbers represented by the tuples $(S^{\Xi}, W^{\Xi}, F^{\Xi})$ and $(S^{\Upsilon}, W^{\Upsilon}, F^{\Upsilon})$, respectively. Here, S indicates the sign, W corresponds to the length of the binary bit string, and F denotes the number of bits used for the fractional part. First, the input Ξ passes through an interval selector unit determining the sub-interval containing Ξ and consequently, the values of parameters specific to the sub-interval e.g., the spacing between breakpoints. Since the interval selector unit is implemented by using a comparator in each node of the binary tree generated from the set of sub-intervals, a single cycle implementation is not appropriate. Moreover, the sequential segmentation approach even generates an unbalanced binary tree, resulting in a generally larger set of cascaded comparators than the other two segmentation approaches. In our design flow, a pre-processing balancing step is therefore applied for any set P of intervals that always delivers a balanced binary tree of comparators. Then, the address generator determines the addresses of the stored function values A_i and A_{i+1} corresponding to the breakpoints enclosing the input. These values are read from the BRAM. In the last stage, the subsequent block performs a linear interpolation to determine Υ.

We implemented a design flow that automates the generation of the shown hardware architecture irrespective of the given function and interval. First, one of the proposed interval splitting algorithms is selected and applied (see Sec. 4). The output P of the algorithm is then used to generate a hardware description in VHDL. For the determination of the range values y_i to be stored in BRAMs, we employ the HDL coder of Matlab [9] and adapt the code generation to instantiate BRAMs. The set P is also directly used to implement the interval selector and the linear interpolation blocks. The arithmetic operations performed to compute the output are pipelined to increase the throughput of the circuit. The interval selector and address generator take three clock cycles together to generate valid address signals. The y values are obtained from the BRAMs in the next clock cycle. Then, the

[1]The minimization of the number of intervals as shown in Fig. 6b is only a secondary goal as we will see when looking at the hardware implementation costs and evaluations in Sects. 5 and 6, particularly Sec. 6.2.2.

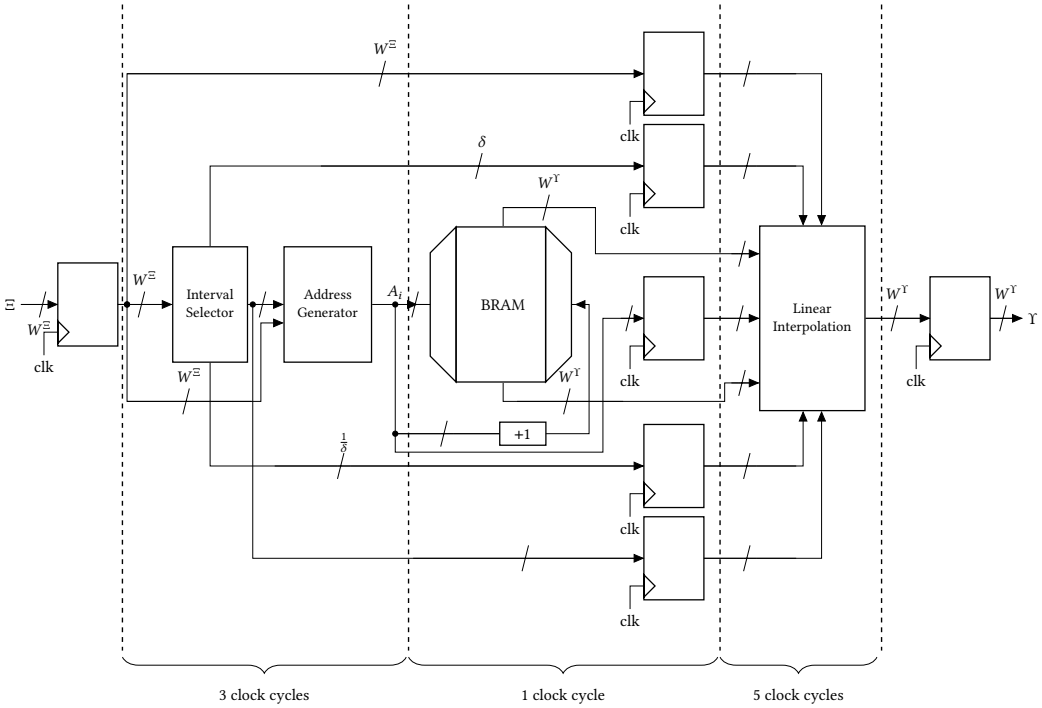


Fig. 7. Proposed generic hardware implementation for table-based function approximation using interval splitting and BRAM instantiation. An input Ξ of W^{Ξ} bits in pre-specified fixed-point number format is evaluated. In just three clock cycles, the interval selector determines in which partition Ξ is and its respective base address A_i in the BRAM, as well as, a valid address is generated. In the next clock cycle, the two breakpoints required to evaluate Ξ are looked up in the BRAM. Then, in another five clock cycles, the linear interpolation block calculates the approximated value of $f(x)$. The shown implementation is pipelined and has a latency of $L = 9$ clock cycles.

pipelined linear interpolation block requiring five clock cycles produces the final output. Therefore, the latency (L) of the proposed architecture is constant at $L = 9$ per function evaluation. Note that this latency is independent of the function to be approximated, number formats, and number of sub-intervals determined by interval splitting. In the following, we evaluate the implementation of our segmentation approach in a target FPGA device to measure the memory footprint reductions, logic utilization (LUTs), BRAM utilization, and the achievable clock frequency.

6 EXPERIMENTAL RESULTS

To evaluate the memory savings of the interval splitting approaches introduced in Sec. 4 also in real hardware, we compare the equidistant spacing table-based function approximation approach (*Reference*) as introduced in Sec. 3 against our newly introduced interval segmentation approximation approach. For the comparison, we consider the hierarchical approach as this has shown in Sec. 4 to provide highest mean memory footprint reductions while producing no more intervals than the other two introduced approaches. The goal of both the *Reference* approach and *Hierarchical segmentation* is to generate an efficient memory footprint function approximation of $f(x)$ for a given interval $[x_0, x_0 + a)$ while never exceeding an absolute error bound E_a.
Apart from comparing the memory footprint reductions ΔM_F obtained by each approach, we

synthesized hardware implementations of both approaches and compared them regarding also the number of utilized BRAMs, the number of utilized LUTs, and the achievable clock frequency in MHz. Here, we used the Matlab/Simulink LUT Optimizer [10] to obtain the VHDL implementations of the *Reference* approach. For the purpose of comparison, we only customized the code delivered by Matlab's HDL coder to force the instantiation of the tabular function representation in BRAMs instead of using LUTs.
Regarding the *Hierarchical segmentation* approach, we applied our newly introduced design flow (see Sec. 5) that automatically performs VHDL code generation and BRAM instantiation of the table-based function approximation. The benchmarks considered for comparison consists again of the nine test functions as presented in Table 1, including standard mathematical functions as well as multiple non-linear activation functions of Deep Neural Networks (DNNs).

In the following, we will show that our proposed hierarchical segmentation-based approach is able to achieve fundamentally higher memory footprint reductions and at the same time a lower number of BRAMs over the *Reference* approach.

6.1 Test Setup

Our benchmark is composed of nine test functions as presented in Table 1. Each function is evaluated by the *Reference* and the *Hierarchical segmentation* approach with a chosen absolute approximation error $E_a = 9.5367E-07$ over the interval $[x_0, x_0 + a)$ presented in the second column in Table 1. The third and fourth columns in Table 1 show the input $(S^{\Xi}, W^{\Xi}, F^{\Xi})$ and output $(S^{\Upsilon}, W^{\Upsilon}, F^{\Upsilon})$ fixed-point format used to approximate the proposed test functions. Here, S corresponds to the bit used to represent the sign, being 1 for a negative number and 0 for a positive. W is the bit-width, and F is the number of bits used for the fractional part.
As a target FPGA, we selected a Zynq-7000 Programmable System-on-Chip (PSoC) with 53,200 LUTs, 106,400 Flip-Flops, and up to 4.9 MB of BRAMs. We performed the synthesis of the circuits for the *Reference* and *Hierarchical segmentation* approaches using Vivado 2021.2. We synthesized nine different circuits per approximated function for the hierarchical approach by varying the number of generated intervals $1 \leq n < 30$, where $n = |P| - 1$ stands for the number of generated intervals. For the trivial case of $n = 1$ (no splitting is performed), the results are equal to those delivered by the *Reference* approach.

6.2 Analysis of Synthesis Results

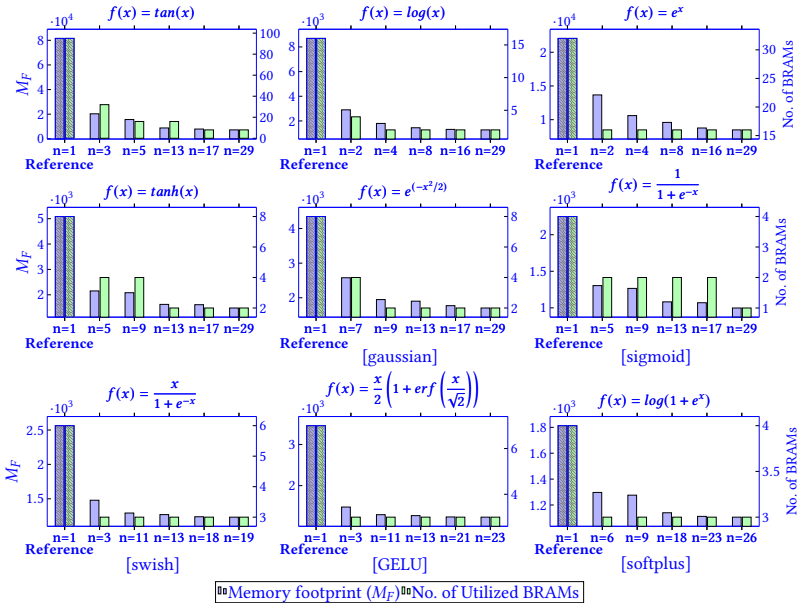
Fig. 8 presents the synthesis results obtained by the *Reference* and our approach *Hierarchical segmentation* regarding memory footprint (M_F), number of instantiated BRAMS, number of utilized LUTs, and achievable clock frequency in MHz.
As can be seen, our proposed hierarchical approach is able to drastically reduce the memory footprint resulting in very efficient utilization of BRAMs (see Fig. 8a). Regarding logic utilization of the target FPGA, our hierarchical approach utilizes insignificantly more LUTs than the reference approach. This increase is due to the number of generated intervals impacting the size and depth of the binary tree of comparators of the interval selector (see Sec. 4). However, this typically represents only a 3 % overall utilization of the LUTs available in the target FPGA. Finally, our proposed approach delivers circuits ranging between 86.5 MHz and 88.5 MHz in terms of achievable clock frequency (see Fig. 8b).

6.2.1 Memory Footprint and BRAMs Utilization. Fig. 8a presents the memory footprint, and the number of utilized BRAMs colored in blue and green, respectively, for the *Reference* (n=1) and the *Hierarchical segmentation* approaches. Here, we generated six implementations using the hierarchical approach for a varying number of generated intervals n for each explored function.

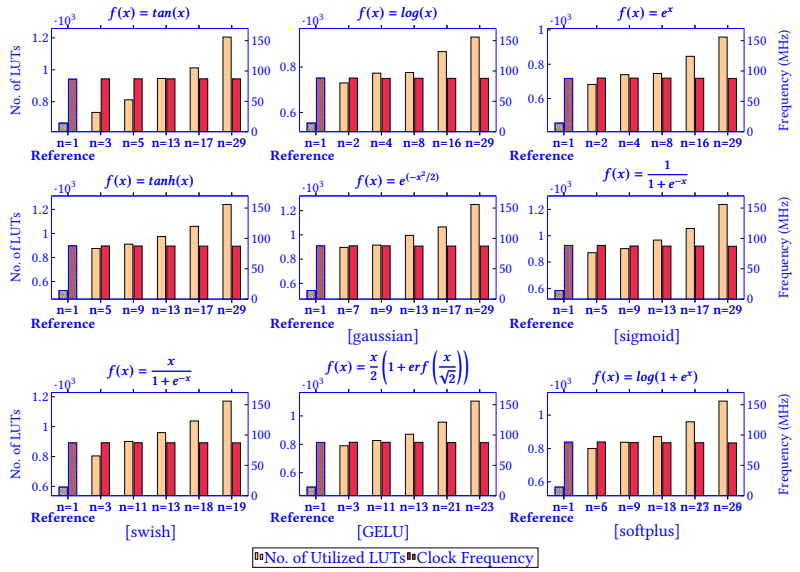
Table 1. Evaluation benchmark composed of nine functions and their characteristics. Column five indicates n, the number of intervals the domain of the function is split. Columns six to eight present the memory footprint reduction ΔM_F, the BRAM reduction $\Delta BRAM$ and the increment of LUTs $\Delta LUTs$ compared to the *Reference* approach in %.

$f(x)$	$[x_0, x_0 + a)$	$(S^{\Xi}, W^{\Xi}, F^{\Xi})$	$(S^{\Upsilon}, W^{\Upsilon}, F^{\Upsilon})$	n	ΔM_F [%]	$\Delta BRAM$ [%]	$\Delta LUTs$ [%]
$tan(x)$	$[-1.5, 1.5)$	$(1, 32, 30)$	$(1, 32, 27)$	3	75 %	66 %	10 %
				5	80 %	83 %	21 %
				13	89 %	83 %	42 %
				17	90 %	91 %	52 %
				29	91 %	91 %	81 %
$log(x)$	$[0.625, 15.625)$	$(0, 32, 28)$	$(1, 32, 29)$	2	66 %	75 %	32 %
				4	79 %	87.5 %	39 %
				8	83 %	87.5 %	40 %
				16	84 %	87.5 %	57 %
				29	85 %	87.5 %	68 %
e^x	$[0, 5)$	$(0, 32, 29)$	$(0, 32, 24)$	2	38 %	50 %	49 %
				4	51 %	50 %	61 %
				8	56 %	50 %	63 %
				16	60 %	50 %	85 %
				29	61 %	50 %	109 %
$tanh(x)$	$[-8, 8)$	$(1, 32, 27)$	$(1, 32, 31)$	5	57 %	50 %	66 %
				9	59 %	50 %	73 %
				13	68 %	75 %	85 %
				17	68 %	75 %	102 %
				29	70 %	75 %	136 %
$e^{(-x^2/2)}$ [gaussian]	$[-6, 6)$	$(1, 32, 28)$	$(1, 32, 32)$	7	40 %	50 %	66 %
				9	55 %	75 %	69 %
				13	56 %	75 %	84 %
				17	59 %	75 %	97 %
				29	60 %	75 %	131 %
$\frac{1}{1+e^{-x}}$ [sigmoid]	$[-10, 10)$	$(1, 32, 27)$	$(0, 32, 32)$	5	42 %	50 %	49 %
				9	43 %	50 %	54 %
				13	51 %	50 %	65 %
				17	52 %	50 %	81 %
				29	55 %	75 %	111 %
$\frac{x}{1+e^{-x}}$ [swish]	$[-5, 5)$	$(1, 32, 27)$	$(0, 32, 32)$	3	42 %	50 %	35 %
				11	49 %	50 %	51 %
				13	50 %	50 %	61 %
				18	51 %	50 %	74 %
				19	52 %	50 %	95 %
$\frac{x}{2}\left(1 + erf\left(\frac{x}{\sqrt{2}}\right)\right)$ [GELU]	$[-5, 5)$	$(1, 32, 27)$	$(0, 32, 32)$	3	58 %	57 %	58 %
				11	63 %	57 %	65 %
				13	64 %	57 %	74 %
				21	65 %	57 %	81 %
				23	65 %	57 %	101 %
$log(1 + e^x)$ [softplus]	$[-5, 5)$	$(1, 32, 27)$	$(0, 32, 32)$	5	28 %	25 %	40 %
				9	29 %	25 %	47 %
				13	37 %	25 %	53 %
				17	38 %	25 %	69 %
				29	39 %	25 %	90 %

For example, when approximating the function $f(x) = tan(x)$, we generated six implementations with $n \in \{1, 3, 5, 13, 17, 29\}$ intervals each. The calculation of the memory footprint obtained by the *Reference* approach ($n = 1$) is according to Eq. (12) and Eq. (13) for *Hierarchical segmentation*.
At first, we can observe a considerable decrease in the memory footprint and the number of used BRAMs when we apply the hierarchical segmentation approach ($n > 1$) for all six considered test functions. The sixth and the seventh columns in Table 1 present the memory footprint (ΔM_F) and the reduction in utilized BRAMs ($\Delta BRAM$) obtained by our approach *Hierarchical segmentation* compared to *Reference* for each obtained partition. The memory footprint reduction ΔM_F was



(a) Memory footprint and BRAM utilization



(b) LUT Utilization and Clock Frequency

Fig. 8. Synthesis results obtained using our *hierarchical segmentation* approach for the approximation of the six benchmark functions in Table 1 against the number of generated sub-intervals ($1 \leq n < 30$). In (a), we can observe that as the number of sub-intervals n increases, so do the reductions in the memory footprint and the number of utilized BRAMs. In (b), the number of utilized LUTs and clock frequency are presented. Here, the number n of intervals affects the number of utilized LUTs which are used to implement the interval selector in Fig. 7. Finally, we can observe that the achieved clock frequency is almost constant at $\sim$ 87 MHz.

calculated according to Eq. (14).
In general, we can observe that as the number n of sub-intervals increases, the memory footprint decreases as well as the number of utilized BRAMs. E.g., for $f(x) = tan(x)$, the *Reference* results in a table with a $M_F^R = 81,543$ entries stored in 95 allocated BRAMs. On the other hand, the table obtained by our approach for a partition with $n = 3$ intervals resulted in a memory footprint reduction $\Delta M_F = 75\,\%$ and BRAM reduction $\Delta BRAM = 66\,\%$. For $n = 5$, the corresponding memory footprint and BRAMs reduction resulted in $\Delta M_F = 80\,\%$ and $\Delta BRAM = 83\,\%$, respectively. However, for $n = 13$, the memory footprint reduced by $\Delta M_F = 89\,\%$ but the $\Delta BRAM = 83\,\%$ remained the same as for $n = 5$. Intuitively, we would expect a reduction in the BRAM usage as the memory footprint reduces. Nevertheless, this might not always be the case because of the storage capacity of each BRAM and how the data is internally stored as illustrated next.
According to the Xilinx 7-series specification, it is important to note that each BRAM can store up to 1,024 entries for an output bit-width of $W = 32$ bits. The number of address bits of one BRAM is thus 10. Similarly, for any memory footprint M_F, the number of required address bits is $\lceil log_2(M_F) \rceil$. Hence, the number of BRAMs of depth 1,024 (2^{10}) required to store M_F values is given by $\frac{2^{\lceil log_2 M_F \rceil}}{1024} = 2^{\lceil log_2 M_F \rceil - 10}$. For example, let's consider two circuits synthesized to approximate $f(x) = tan(x)$ for $n = 5$ and $n = 13$ intervals. The reported memory footprints are $M_F = 15,644$ and $M_F = 8,798$, respectively. The number of address bits for both the cases is 14 and therefore, the number of allocated BRAMs is $2^{14-10} = 16$ for both implementations despite the large difference in memory footprints.
Also for the other test functions, we obtained significant memory footprint reductions and efficient BRAM utilizations. For example, for $f(x) = log(x)$, we reported memory footprint reductions ranging from 66 % to 85 % and BRAMs reduction ranging from 75 % to 87.5 %. In case of $f(x) = e^x$, the memory footprint reductions ΔM_F ranged from 38 % to 61 % with a BRAMs reduction $\Delta BRAM$ of 50 %. For $f(x) = tan(x)$, the ΔM_F ranged from 57 % to 70 % with a BRAMs reduction $\Delta BRAM$ up to 75 %. For $f(x) = e^{-\frac{x^2}{2}}$, the ΔM_F ranged from 40 % to 60 % with a BRAMs reduction $\Delta BRAM$ up to 75 %. Finally, for $f(x) = \frac{1}{1+e^{-x}}$, the ΔM_F ranged from 42 % to 55 % with a BRAMs reduction $\Delta BRAM$ of up to 75 %.

6.2.2 LUT Utilization. In Fig. 8b, the light orange bars present the number of LUTs utilized for the six implementations obtained by the proposed *Hierarchical segmentation* approach ranging the number of generated intervals from $n = 1$ to 30. Here, the *Reference* corresponds to the case with no partition ($n = 1$). We can observe that the number of utilized LUTs increases with the number of generated intervals n for all the presented benchmark functions. The increment in LUTs can be attributed to an increase in the number of comparisons required to traverse the binary tree in the interval selection block (see Fig. 7), which are tightly related to the number of generated intervals n obtained by our proposed hierarchical approach. However, for all values n of intervals, the absolute overall overhead is typically less than 3 % of the available LUTs in the target FPGA.

6.2.3 Clock Frequency. In Fig. 8b, the red bars show the clock frequency achieved by six implementations of the proposed *hierarchical segmentation* approach by varying the number of generated intervals n between 1 to 30. The architecture is fully pipelined with a data introduction interval of only one clock cycle to start subsequent function evaluations. We can generally observe that the clock frequency lies between 86.5 MHz to 88.5 MHz for the six analyzed functions. For all test functions, the critical path lies in the linear interpolation unit. A multiplication is carried out by two cascaded DSPs (Digital Signal Processor) which introduce an almost constant delay for all implementations. Slight differences (typically less than 2 MHz) are caused just by minor variations in the net and routing delays from design to design. Our proposed hierarchical approach reaches

an overall average clock frequency of 87.5 MHz. Accordingly, the hardware implementation is able to produce an approximated function evaluation within $\frac{9\ clock\ cycles}{87.5\ MHz} \approx 102.8\ ns$.

7 RELATED WORK

As our proposed approach can be utilized for approximating basically any given elementary mathematical function $f(x)$ in a given interval and given a maximum absolute tolerable error E_a, we first compare our approach to state-of-the-art methods proposing hardware circuit implementations for the approximated evaluation of elementary functions [10, 15–23]. In general, approaches for computing elementary functions can be classified into three categories: 1) polynomial-based, 2) iterative approximation algorithms, and 3) table-based approaches. These approaches are able to trade off accuracy, latency, area, and memory footprint.

Polynomial-based approaches [15–20] evaluate polynomial functions for function approximation including linear regression, polynomials of low degree or spline functions. Although resulting circuit implementations often require only a negligible memory footprint, the evaluations including multiplications and additions can be quite costly in terms of resources and evaluation times, particularly if low approximation error bounds are given. Thus, polynomial-based approaches become impractical in environments where low circuit area, low latency and low energy margins are crucial.

Iterative approximation algorithms, as the name suggests, calculate a function based on iterative evaluations. Despite being resource-efficient, these algorithms typically suffer from slow convergence, thus requiring many iterations to achieve a specific approximation error. One prominent representative of this evaluator type is CORDIC algorithms [21, 22], which approximate trigonometric and hyperbolic functions. However, the high latency of iterative approaches might imply an energy overhead not desired or tolerable in mobile or edge devices [24, 25].

Table-based approaches [10, 23] split a given function interval into discrete points called breakpoints and store the function values evaluated at these breakpoints in a lookup table. Although offering constant-time lookup, the main drawback is often the memory footprint in terms of the size of the resulting tables, that grows exponentially with the bit-width of the input. Hence, table-based methods are often combined with polynomial-based approximation between stored breakpoints to reduce the memory footprint. In the area of polynomial-based approximation, particularly piecewise-polynomial approximations [17–20] were proposed. The approaches in [18] and [20] segment a given interval to be approximated using piece-wise linear interpolations of the form $ax+b$. Both approaches produce gradient-based non-uniform segments such that the approximation error in each segment does not exceed a specified maximal error. But a typically large number of segments determines the number of comparisons performed to place a given input into the correct segment and subsequently perform a piece-wise linear interpolation. Both of these numbers increase with the input interval length and the steepness of the function.

Contrary to all of these approaches, our proposed interval splitting algorithms (1) segment a given domain of a function (interval) into typically a small set of sub-intervals by using *gradient information*. Thereby, the number of sub-intervals created can be controlled by a threshold parameter ω. (2) Inside each sub-interval, no interpolation is applied, but rather an even sampling. (3) By construction, a maximal approximation error bound E_a is guarded over the whole function domain. (4) To reduce the number of breakpoints to be stored within each interval, we finally exploit a piece-wise linear interpolation similar to [18, 20]. But even in this single step common to [18, 20], we do not have to explicitly store any slope a and offset b, but only the values y_i and y_{i+1} of the nearest breakpoints x_i and x_{i+1} which are obtained by a simple memory lookup and constant time of just 5 clock cycles per function evaluation.

Finally, our approach might be a great candidate also for the low-cost and at the same time low-latency evaluation of activation functions in ANNs and DNNs on edge devices. In the following, we therefore compare our approach to methods that propose custom hardware implementations of activation functions to be utilized to either accelerate the training and inference phases of a neural network or to implement low-power approximate activation function evaluators [26–29].

Approximation of activation functions for ANNs on edge and low-power devices has become of interest due to the complexity of state-of-the-art DNNs. As the number of input parameters of state-of-the-art DNNs grows [30, 31], the number of operations, such as matrix multiplications and activation function evaluations in the training and inference phases, drastically increases, too. Consequently, the search for alternative hardware architectures for providing low resource cost and low energy consuming circuit implementations for DNNs has become crucial. In this context, FPGA devices to implement custom designs of activation functions have been explored as a low-power alternative [26–29]. Also, approximate computing techniques have been utilized to further optimize the energy consumption of those hardware implementations. E.g., [27] presents a custom combinatorial hardware implementation that calculates hyperbolic tangent, ReLU, and sigmoid functions in the same circuit. For instance, the approach divides the hyperbolic tangent domain into three intervals. From $(0, 1]$, the function is approximated as a linear function, and from $[2, 8)$, the approximated output is 1. A table is utilized to evaluate hyperbolic tangent in the range of $(1, 2)$. The circuit has 3 bits as input and 5 bits as output, resulting in an equidistant sampled table of 8 entries, each 5 bit in length. Unfortunately, the approach is fully specific to the function and is also not able to satisfy a given maximum absolute error bound E_a as our approach. For the same hyperbolic function, [29] suggests to reduce the function approximation problem to an m-output Boolean circuit synthesis problem. For $n = 4$ considered input bits and $m = 7$ output bits, the function approximator circuit then consists of 7 digital circuits and avoids any function tables. However, such an approach becomes infeasible for functions requiring low approximation error margins and easily explodes in hardware cost for real-world input and output word lengths (e.g., $n = m = 32$ as considered in all our test functions in Table 1 and circuits generated). Only the two approaches presented in [28] and [32] resemble our interval splitting approach to some extent: [28] proposes a table-based approach for approximating DNN activation functions. An offline-trained neural network determines a set of typically not equidistant breakpoints to be stored. As neural networks can also be seen as function approximators, the proposed approach consists first in training a one-layer fully connected neural network to predict a given function. The neurons and weights of the trained neural network determine an approximation of the target function by a set of linear functions which are then stored in a LUT-based table to evaluate the function $f(x)$. However, the approach is first not only very computationally intensive due to a required neural network training, but second also not able to guarantee a given maximum approximation error bound E_a by construction. Third, our interval splitting approach does not determine breakpoints but rather intervals in which equidistant sampling is performed to determine the breakpoints. The approach [28] might not scale for low error margins as considered in this paper.

Last, we can find approaches that propose a partitioning of the domain of a function into sub-intervals as our proposed approach. As an example, [32] performs an approximation of additions and multiplications on accelerators for K-means and Support-Vector Machine (SVM). A two dimensional table is proposed to store the output of a multiplication given two input operands. Different quantization levels in the breakpoints stored can be exploited to reduce the table size using an input-aware approximation consisting of learning the partitioning of the domain. Here, a training dataset is utilized to generate a partitioning while considering the statistics of the inputs by calculating the probability distribution of the input dataset elements. Instead of uniformly partitioning, the partition is determined by the probability of the elements in the training dataset. Contrary to our

approach which does not require any input dataset to determine an interval partition, the approach in [32] is also not able to guarantee any maximal approximation error over the domain of the function.

In this paper, we have shown that the proposed idea of interval splitting can help to reduce the memory footprint of table-based function approximation drastically without sacrificing a given approximation error bound. In contrast to custom circuits for evaluating activation functions, our approach guarantees not to exceed a user-given maximum absolute approximation error bound E_a over the specified domain of a function and is fully automatic in synthesizing a low cost and low latency table-based circuit implementation. It has been shown that our approach is flexible for the evaluation of a large scope of elementary but also several DNN activation functions in just nine clock cycles per evaluation.

8 CONCLUSION

In this article, we investigated an efficient way to approximate elementary functions in a given domain (interval) and given a maximum absolute error using interval splitting. First, we realized that a huge reduction in memory footprint can be obtained over storing a function table with uniform sampling by splitting the given domain into a set of sub-intervals and assuming a coarser sampling grid for low gradient regions, while always respecting a given maximum approximation error E_a. Second, we proposed a generic hardware architecture that synthesize such interval-splitted tabular function approximators with a constant and function-independent evaluation latency of just $L = 9$ clock cycles per function evaluation. Moreover, we exploited explicitly the use of BRAMs by automatically inferring them in our design flow during the code generation of the hardware description. In consequence, huge reductions in the memory footprint were shown to be achievable by our proposed approach. Thus, the usage of BRAMs allows to save huge amounts of LUTs that can be used better to implement other logic functions. The presented results inspire to investigate the exploration also of energy and power tradeoffs in dependence of the accuracy as part of future work. For instance, in some realms, e.g., mobile computing and edge computing, a lower power consumption might be required at the expense of less accuracy [12, 24, 25]. With our presented approach, it should be possible to explore these tradeoffs in future work as the maximum tolerable error as well as the input and output word sizes are parameters that can be tuned.

REFERENCES

[1] Q. Xu, T. Mytkowicz, and N. Kim. Approximate computing: A survey. *IEEE Design & Test*, 33(1):8–22, 2015.
[2] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*, pages 1–6, 2013.
[3] A. Becher, J. Echavarria, D. Ziener, S. Wildermann, and J. Teich. A LUT-Based Approximate Adder. In *Proceedings of FCCM 2016*, pages 27–27, 2016.
[4] Hyoju Seo, Yoon Seok Yang, and Yongtae Kim. Design and analysis of an approximate adder with hybrid error reduction. *Electronics*, 9(3), 2020.
[5] Jorge Echavarria, Stefan Wildermann, Andreas Becher, Jürgen Teich, and Daniel Ziener. FAU: Fast and Error-optimized Approximate Adder Units on LUT-Based FPGAs. In *Proceedings of Field-Programmable Technology (FPT)*, pages 213–216, 2016.
[6] Srinivasan Narayanamoorthy, Hadi Asghari Moghaddam, Zhenhong Liu, Taejoon Park, and Nam Sung Kim. Energy-efficient approximate multiplication for digital signal processing and classification applications. *IEEE Transactions on Very Large Scale Integration Systems*, 23(6):1180–1184, 2015.
[7] Mohammad Ahmadinejad, Mohammad Hossein Moaiyeri, and Farnaz Sabetzadeh. Energy and area efficient imprecise compressors for approximate multiplication at nanoscale. *International Journal of Electronics and Communications*, 110:152859, 2019.
[8] M. Brand, M. Witterauf, F. Hannig, and J. Teich. Anytime instructions for programmable accuracy floating-point arithmetic. In *Proceedings of ACM Int. Conf. on Comp. Fronts. (CF), 2019*, pages 215–219, 2019.
[9] MATLAB. *version 9.6.0 (R2019a)*. The MathWorks Inc., Natick, Massachusetts, 2019.

[10] MATLAB. *Optimize Lookup Tables for Memory-Efficiency Programmatically*. The MathWorks Inc., 2021.

[11] Xilinx. *7 Series FPGA Memory Resources*. https://www.xilinx.com/, 2019.

[12] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv.*, 51(5), sep 2018.

[13] Md Zahangir Alom, Tarek M. Taha, Chris Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Mahmudul Hasan, Brian C. Van Essen, Abdul A. S. Awwal, and Vijayan K. Asari. A state-of-the-art survey on deep learning theory and architectures. *Electronics*, 8(3), 2019.

[14] Jean-Michel Muller. A few results on table-based methods. *Reliable Computing*, 5:279–288, 1999.

[15] Ben Adcock, Simone Brugiapaglia, and Clayton G. Webster. *Compressed Sensing Approaches for Polynomial Approximation of High-Dimensional Functions*, pages 93–124. Springer International Publishing, 2017.

[16] David I Shuman, Pierre Vandergheynst, and Pascal Frossard. Chebyshev polynomial approximation for distributed signal processing. In *2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*, pages 1–8, 2011.

[17] Dong-U Lee, Ray C. C. Cheung, Wayne Luk, and John D. Villasenor. Hierarchical segmentation for hardware function evaluation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(1):103–116, 2009.

[18] Jon T. Butler, C.L. Frenzen, Njuguna Macaria, and Tsutomu Sasao. A fast segmentation algorithm for piecewise polynomial numeric function generators. *Journal of Computational and Applied Mathematics*, 235(14):4076–4082, 2011.

[19] Davide De Caro, Nicola Petra, and Antonio G. M. Strollo. Efficient logarithmic converters for digital signal processing applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 58(10):667–671, 2011.

[20] Hongxi Dong, Manzhen Wang, Yuanyong Luo, Muhan Zheng, Mengyu An, Yajun Ha, and Hongbing Pan. Plac: Piecewise linear approximation computation for all nonlinear unary functions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(9):2014–2027, 2020.

[21] Ray Andraka. A survey of cordic algorithms for fpga based computers. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, FPGA '98, page 191–200. Association for Computing Machinery, 1998.

[22] Linbin Chen, Jie Han, Weiqiang Liu, and Fabrizio Lombardi. Algorithm and design of a fully parallel approximate coordinate rotation digital computer (cordic). *IEEE Transactions on Multi-Scale Computing Systems*, 3(3):139–151, 2017.

[23] Ye Tian, Ting Wang, Qian Zhang, and Qiang Xu. Approxlut: A novel approximate lookup table-based accelerator. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 438–443, 2017.

[24] Linghe Kong, Jinlin Tan, Junqin Huang, Guihai Chen, Shuaitian Wang, Xi Jin, Peng Zeng, Muhammad K. Khan, and Sajal K. Das. Edge-computing-driven internet of things: A survey. *ACM Comput. Surv.*, aug 2022. Just Accepted.

[25] Kaoru Ota, Minh Son Dao, Vasileios Mezaris, and Francesco G. B. De Natale. Deep learning for mobile multimedia: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 13(3s), jun 2017.

[26] Shi Dong, Ping Wang, and Khushnood Abbas. A survey on deep learning and its applications. *Computer Science Review*, 40:100379, 2021.

[27] Chih-Hsiang Chang, Hsu-Yu Kao, and Shih-Hsu Huang. Hardware implementation for multiple activation functions. In *2019 IEEE International Conference on Consumer Electronics - Taiwan (ICCE-TW)*, pages 1–2, 2019.

[28] Joonsang Yu, Junki Park, Seongmin Park, Minsoo Kim, Sihwa Lee, Dong Hyun Lee, and Jungwook Choi. Nn-lut: Neural approximation of non-linear operations for efficient transformer inference. In *Proceedings of Design Automation Conference (DAC)*, DAC '22, page 577–582, New York, NY, USA, 2022. Association for Computing Machinery.

[29] Tao Yang, Yadong Wei, Zhijun Tu, Haolun Zeng, Michel A. Kinsy, Nanning Zheng, and Pengju Ren. Design space exploration of neural network activation function circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(10):1974–1978, 2019.

[30] Ross Gruetzemacher and David Paradice. Deep transfer learning and beyond: Transformer language models in information systems research. *ACM Comput. Surv.*, 54(10s), sep 2022.

[31] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[32] Arnab Raha and Vijay Raghunathan. Qlut: Input-aware quantized table lookup for energy-efficient approximate accelerators. *ACM Transactions on Embedded Computing Systems*, 16(5s), sep 2017.