Efficient Integer-Only Implementation of Tanh and Sigmoid for Embedded AI on RISC-V

Kamil Kaczmarski[1 [0009−0000−6870−2373]], Pawel Gepner[1 [0000−0003−0004−1729]], Ewa Deelman[2 [0000−0001−5106−503X]], Pawel Poczekajlo[3 [0000−0003−4742−7872]], Leonid Moroz[1 [0000−0003−4131−309X]], and Nataliia Gavkalova[1 [0000−0003−1208−9607]]

[1] Warsaw University of Technology, Warsaw, Poland
Faculty of Mechanical and Industrial Engineering
{kamil.kaczmarski.dokt,pawel.gepner,leonid.moroz,nataliia.gavkalova}@pw.edu.pl
[2] University of Southern California, Marina del Rey CA 90292, USA
USC Information Sciences Institute
deelman@isi.edu
[3] Koszalin University of Technology, Koszalin, Poland
Faculty of Electronics and Computer Science
pawel.poczekajlo@tu.koszalin.pl

Abstract. We present a unified integer-only kernel for hyperbolic functions `tanh(x)` and `sigmoid(x)`, designed for embedded RISC-V platforms without floating-point units. The kernel combines LUT anchoring, CORDIC microrotations, and linear correction in Q20 arithmetic, producing outputs directly in Q1.16 format suitable for ML inference. Exhaustive evaluation across the entire 17-bit input space demonstrates deterministic accuracy better than 1 ULP in Q1.16, with maximum absolute errors below 1.5×10^{-5}. Compared to software-emulated floating-point implementations, our approach achieves significant speedup while reducing hardware complexity by unifying multiple activation functions in a single IP block. This makes the method highly relevant for efficient deployment of neural networks on resource-constrained RISC-V microcontrollers.

Keywords: CORDIC · integer-only · ML inference · RISC-V · sigmoid.

1 Introduction

Artificial intelligence (AI) computations are increasingly being executed on ultra-low-power devices, where energy efficiency and low latency are critical. In this context, the open RISC-V architecture is gaining significant traction due to its extensibility and support for specialized extensions, including reduced-precision floating-point formats such as BFloat16 [16,3].

At the same time, reduced-precision numerical formats have become an important component of modern AI and machine learning (ML) systems. Formats

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

such as BFLOAT16 are widely adopted in training and inference pipelines because they preserve much of the dynamic range of FP32 while reducing memory usage and improving computational throughput. Previous studies have shown that neural networks trained and evaluated using BFLOAT16 can achieve accuracy comparable to FP32 baselines across a wide range of architectures and tasks, confirming that reduced precision is often sufficient for practical AI workloads [11,19,2,10]. These trends are reflected in both academic research and industry practice, where reduced-precision computation is now standard in many deep learning frameworks and accelerators, including systems targeting embedded and low-power deployments [10,2]. However, on resource-constrained RISC-V microcontrollers lacking hardware floating-point units, even reduced-precision floating-point arithmetic may still require software emulation and therefore incur significant overhead. This gap between the algorithmic suitability of reduced precision and the practical cost of software-based floating-point execution motivates the exploration of integer-only alternatives for nonlinear activation functions used in neural network inference. Although BFLOAT16 helps motivate the broader reduced-precision context of this work, it is not used in the proposed implementation or in the experimental evaluation reported in Section 4.

Many RISC-V microcontrollers lack on-board FPU hardware. As a result, floating-point operations must be performed in software using libraries such as SoftFloat [7], introducing substantial performance overhead. This issue is particularly important for nonlinear activation functions such as `tanh` and `sigmoid`, which rely on computationally expensive hyperbolic or exponential operations [13].

To overcome these limitations, this paper proposes a unified integer-only activation kernel based on CORDIC-derived transformations and fixed-point arithmetic. The kernel efficiently computes both `tanh` and `sigmoid` using a single implementation, addressing an important gap in current RISC-V software support for AI inference and ML applications at the edge.

2 Related Work

Existing research on acceleration of nonlinear functions and lightweight arithmetic techniques for embedded systems highlights several important trends. Reduced-precision floating-point formats such as BFloat16 have been proposed to improve inference efficiency in RISC-V systems [16,3]. Studies evaluating BFloat16 arithmetic report that its reduced mantissa precision does not significantly affect the accuracy of modern neural networks, with models trained using BFloat16 achieving accuracy comparable to FP32 baselines across multiple tasks [11,19]. However, on platforms lacking an FPU, floating-point operations must still be emulated in software using libraries such as SoftFloat [7], which significantly degrades performance on microcontrollers.

Fixed-point arithmetic is a long-standing technique in embedded systems, and numerous frameworks provide efficient implementations of linear operations. For example, the RISC-V Vector Math Library [17] demonstrates continued

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

ecosystem development. However, existing libraries rarely include integer-only optimized implementations of nonlinear activation functions commonly used in machine learning.

CORDIC algorithms have been widely explored for computing transcendental functions using only shifts and additions. Several works demonstrate CORDIC accelerators implemented for RISC-V SoCs and FPGA-based systems [18], confirming their suitability for resource-constrained platforms. These implementations typically focus on specific functions, whereas this work aims to provide a shared kernel that supports both `tanh` and `sigmoid`.

Additionally, activation-function approximations—including piecewise-linear functions, rational models, and continued-fraction expansions—have been applied to reduce computational cost in embedded ML systems. Previous research highlights the high latency of exponential function evaluation in conventional implementations and offers various simplifications suitable for FPGA or ASIC deployment [9,15,8]. Representative low-precision alternatives include K-tanh [13], which optimizes tanh evaluation for deep-learning workloads, and piecewise-linear sigmoid approximations [15], which reduce computational cost through function-specific segmentation. In contrast, our objective is not to optimize each nonlinearity separately, but to reuse a single integer-only kernel for both `tanh` and `sigmoid` with deterministic latency and a shared fixed-point data path.

Research into approximate arithmetic at the instruction-set level further supports the trend toward simplified computation in edge systems. Studies proposing approximate division and square-root operations for RISC-V [14] demonstrate that reduced precision can yield substantial energy and performance benefits.

While existing research addresses fixed-point arithmetic, CORDIC methods, approximation strategies, and reduced-precision floating-point, no prior work offers a unified, integer-only kernel for both `tanh` and `sigmoid` designed for software execution on resource-constrained RISC-V microcontrollers.

3 Description of the Proposed Method

Compared with standard hyperbolic CORDIC realizations, the proposed unified integer-only kernel simultaneously approximates the hyperbolic functions `tanh(x)` and `sigmoid(x)`, using a combination of LUT anchoring, CORDIC microrotations and a final linear correction. All internal computations are performed in Q20 fixed-point format, while the outputs are suitable for conversion to Q1.16 for neural network inference.

The kernel accepts a 17-bit recoded representation of the input argument x,

$$a = (a_0, a_1, \ldots, a_{16}), \quad a_i \in \{0, 1\},$$

which compactly encodes a hyperbolic angle through a LUT index, a sequence of CORDIC microrotations, and a residual correction term. For clarity, we first describe the structure of this encoding and then the three stages of the kernel.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

3.1 Input Recoding and Dynamic Range

The 17-bit input word is partitioned into three logical fields:

- **Anchor index** a_0, a_1, a_2: the three most significant bits select one of $2^3 = 8$ precomputed anchor points stored in LUTs `xx_h` and `yy_h`. The index is
$$j = 4a_0 + 2a_1 + a_2 \in \{0, \ldots, 7\}.$$
- **CORDIC microrotation control** $a_3, \ldots, a_8$: six bits control the direction of each CORDIC iteration in hyperbolic mode. For iteration $i \in \{3, \ldots, 8\}$, bit a_i decides whether the step is performed in the "positive" or "negative" direction.
- **Residual linear correction** $a_9, \ldots, a_{16}$: the last eight bits encode a residual angle in Q20 format,
$$\theta_3 = \sum_{k=0}^{7} a_{9+k} \cdot w_k,$$
where $w_k \in \{2048, 1024, \ldots, 16\}$ are power-of-two weights corresponding to Q20 fixed-point (i.e., $w_k = 2^{11-k}$). Combined with a small coarse correction term δ (derived from a_3, a_4, a_5, see below), this yields a final residual
$$z = \delta + \theta_3 \in \text{Q20}.$$

The mapping from bits to the real-valued argument x is deterministic and can be written as

$$x = \operatorname{atanh}\left(\frac{y_0}{x_0}\right) + \sum_{i=3}^{8} s_i \operatorname{atanh}\left(2^{-(i+1)}\right) + \frac{z}{2^{20}},$$

where (x_0, y_0) is the selected anchor vector, and $s_i \in \{+1, -1\}$ is the sign implied by bit a_i. This parametrisation implicitly defines the dynamic range of x; in our implementation, it corresponds to the range of arguments for `tanh(x)`and `sigmoid(x)`.

3.2 LUT Anchoring

The first stage of the kernel selects an *anchor point* from a small LUT of precomputed hyperbolic vectors. For each index j we store an integer pair

$$(x_0, y_0) = (\texttt{xx\_h}[j], \texttt{yy\_h}[j]),$$

representing a scaled hyperbolic vector

$$(x_0, y_0) \approx K \cdot 2^{20} \cdot (\cosh(\alpha_j), \sinh(\alpha_j)),$$

for some anchor angle α_j and global scaling factor K (Section 3.6).

The anchors are chosen such that they tile the relevant argument range with relatively coarse steps, and the subsequent CORDIC microrotations and residual correction need to compensate only small deviations from the anchor, which reduces error and iteration count.

At this point we initialize the CORDIC state as

$$x_3 = x_0, \quad y_3 = y_0.$$

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

3.3 Hyperbolic CORDIC Microrotations

The second stage performs a fixed number of hyperbolic CORDIC iterations in vectoring mode, using the integer recursion

$$x_{i+1} = x_i \pm (y_i \gg (i+1)),$$
$$y_{i+1} = y_i \pm (x_i \gg (i+1)),$$

for $i = 3, 4, \ldots, 8$, where $\gg$ denotes an arithmetic right shift. The sign in each iteration is controlled by bit a_i:

- if $a_i = 1$: we use the "positive" direction,
- if $a_i = 0$: we use the "negative" direction.

In real arithmetic, the corresponding transformation is

$$(x_{i+1}, y_{i+1}) \approx (\cosh(\pm\alpha_i)\, x_i + \sinh(\pm\alpha_i)\, y_i,\ \sinh(\pm\alpha_i)\, x_i + \cosh(\pm\alpha_i)\, y_i),$$

where the microrotation angles are

$$\alpha_i = \operatorname{atanh}\left(2^{-(i+1)}\right).$$

The cumulative effect of the LUT anchor and microrotations is

$$(x_9, y_9) \approx K \cdot 2^{20} \cdot (\cosh(\tilde{x}),\ \sinh(\tilde{x})),$$

where

$$\tilde{x} = \alpha_j + \sum_{i=3}^{8} s_i \alpha_i$$

is the intermediate angle before residual correction.

Using only six microrotations keeps the kernel small and deterministic in time, which is crucial for embedded systems and hardware implementations.

3.4 Residual Angle and Linear Post-Rotation

The third stage compensates the residual angular error left after the finite sequence of microrotations. We approximate a small additional angle z (in Q20) by combining:

- a coarse term based on a small set of weights (d_3, d_4, d_5),

$$\delta = a_3 \cdot d_3 + a_4 \cdot d_4 + a_5 \cdot d_5,$$

- and a fine term from the residual bits $a_9, \ldots, a_{16}$,

$$\theta_3 = \sum_{k=0}^{7} a_{9+k} \cdot w_k.$$

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

Together they form

$$z = \delta + \theta_3 \in \text{Q20},$$

which corresponds to a small real-valued angle $z/2^{20}$.

For such small angles, the hyperbolic rotation

$$x' = x_9 \cosh\left(\frac{z}{2^{20}}\right) + y_9 \sinh\left(\frac{z}{2^{20}}\right),$$
$$y' = y_9 \cosh\left(\frac{z}{2^{20}}\right) + x_9 \sinh\left(\frac{z}{2^{20}}\right)$$

can be linearized as

$$\cosh(\epsilon) \approx 1, \quad \sinh(\epsilon) \approx \epsilon$$

for $|\epsilon| \ll 1$. Substituting $\epsilon = z/2^{20}$ yields the integer update:

$$x_{17} = x_9 + \left(\frac{z}{2^{20}} \cdot y_9\right) \approx x_9 + ((z \cdot y_9) \gg 20),$$
$$y_{17} = y_9 + \left(\frac{z}{2^{20}} \cdot x_9\right) \approx y_9 + ((z \cdot x_9) \gg 20),$$

which preserves the Q20 scaling. This yields the final kernel outputs:

$$(x_{17}, y_{17}) \approx K \cdot 2^{20} \cdot (\cosh(x),\ \sinh(x)).$$

The use of a single linear post-rotation instead of additional CORDIC iterations reduces computational cost and hardware complexity while keeping the approximation error below one LSB in Q1.16, as demonstrated in Section 4.3 in Table 3. Our algorithm is presented in the schematic view in Figure 1.

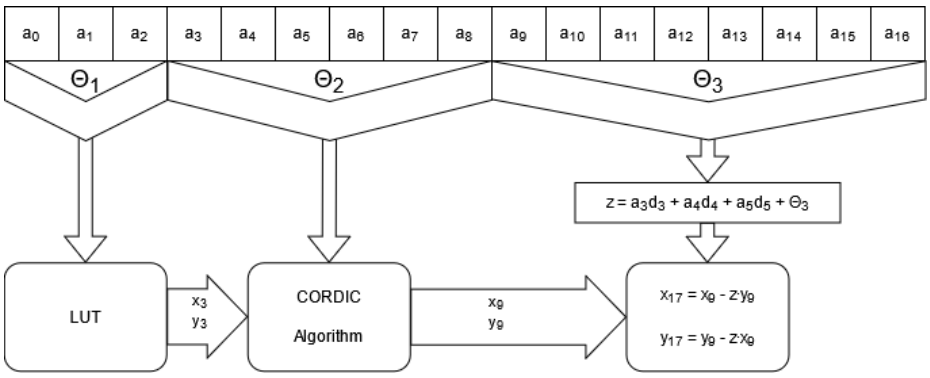

Fig. 1. Schematic view of our CORDIC algorithm.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

3.5 Extraction of Tanh and Sigmoid

Once the final vector (x_{17}, y_{17}) is obtained, we can derive `tanh(x)` and `sigmoid(x)` directly in the integer domain.

From the hyperbolic identity

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)},$$

and the scaling

$$x_{17} \approx K \cdot 2^{20} \cosh(x), \quad y_{17} \approx K \cdot 2^{20} \sinh(x),$$

we obtain

$$\tanh(x) \approx \frac{y_{17}}{x_{17}}.$$

The global scale factor $K \cdot 2^{20}$ cancels exactly, and the ratio can be computed as a fixed-point division producing a Q1.16 result.

For the sigmoid function $sigmoid(x) = 1/(1 + e^{-x})$, we exploit the relation

$$e^{\pm x} = \cosh(x) \pm \sinh(x),$$

so that

$$e^{-x} \approx \frac{x_{17} - y_{17}}{K \cdot 2^{20}}.$$

Substituting into the definition of $sigmoid(x)$ yields

$$sigmoid(x) \approx \frac{1}{1 + \frac{x_{17} - y_{17}}{K \cdot 2^{20}}} = \frac{K \cdot 2^{20}}{K \cdot 2^{20} + x_{17} - y_{17}}.$$

In our implementation, we precompute

$$\texttt{scale} = K \cdot 2^{20},$$

and evaluate

$$sigmoid(x) \approx \frac{\texttt{scale}}{\texttt{scale} + x_{17} - y_{17}}$$

using integer arithmetic followed by a conversion to Q1.16.

Both activations thus share the same kernel (x_{17}, y_{17}), and differ only in a small scalar post-processing step. Figure 2 presents a block diagram for our complete algorithm, to calculate `tanh(x)` and `sigmoid(x)`.

3.6 Fixed-Point Formats and Scaling Factor

All intermediate values (x_i, y_i), the residual z, and the LUT entries are represented in Q20 fixed-point format. The global scaling factor K is chosen once during design time to:

– minimize the worst-case approximation error over the target input range,

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

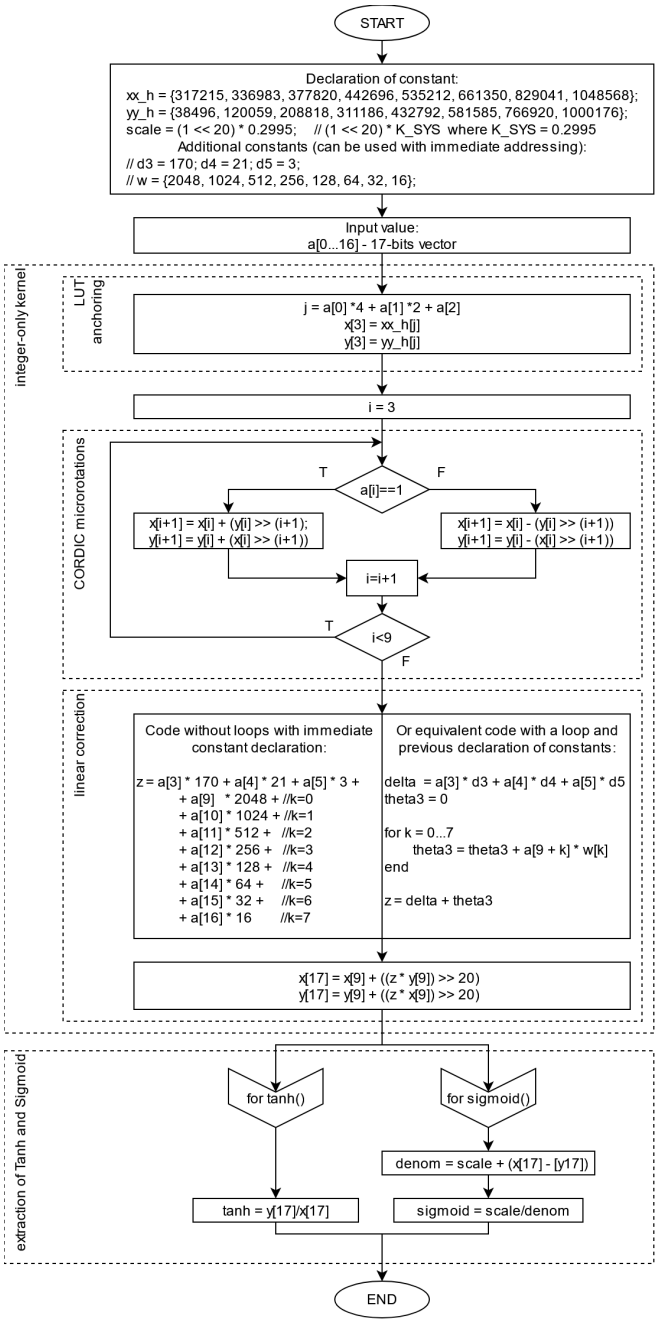

Fig. 2. Diagram showing the implementation scheme of the presented algorithm for `tanh(x)` and `sigmoid(x)`.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

– avoid overflow in the CORDIC iterations and post-rotation,
– and provide a convenient mapping to Q1.16 at the output.

In practice, we use a constant K_{SYS} (stored as a floating-point value only in the reference model; the hardware implementation uses an integer equivalent). The final Q20 outputs (x_{17}, y_{17}) can be converted to Q1.16 by a right shift and optional rounding:

$$v_{\mathrm{Q1.16}} = \mathrm{clip}\left(\lfloor v_{\mathrm{Q20}} \gg 4 \rceil\right),$$

where the clip operation enforces the range $[-1, 1]$ for `tanh` and $[0, 1]$ for `sigmoid`.

In the final fixed-point implementation, the sigmoid denominator is evaluated using an equivalent scaled form, with $\texttt{scale} = K \cdot 2^{21}$ and $((x17 - y17) << 1)$, in order to preserve Q1.16 accuracy after conversion. This allowed us to avoid specific errors caused by quantizing the `scale` factor. In this way, the output value for sigmoid() is calculated correctly and in accordance with the previously indicated precision.

This fixed-point setup leads to deterministic, integer-only evaluation of hyperbolic activations, which we analyze in terms of accuracy and performance in the following sections.

4 Implementation and Results

The implementation was carried out on a microcontroller based on the RISC-V core. The ESP32-C3FH4 chip from Espressif was selected (due to its high popularity and easy availability). It is a highly integrated SoC device. The microprocessor provides hardware support for WiFi and Bluetooth wireless communication. This makes it popular for use in IoT and SmartHome devices. The ESP32 C series chips are Ultra-Low Power solutions. The chip is a complete integrated operating system, eliminating the need for external memory, for example. This startup procedure greatly facilitates implementation and testing. The manufacturer provides for the possibility of running and operating the chip under the control of a dedicated FreeRTOS system. The programs can be written in MicroPython or C compatible with the RISCV32 GCC compiler (riscv32-unknown-elf-gcc). For compiling GCC-compatible code, the operating system (FreeRTOS) can be limited to the minimum configuration necessary for the core and peripheral components (e.g., interfaces, timers) to function properly. Basic parameters of the ESP32-C3FH4 are:

– Single Core RISC-V CPU (32 bits, 160 MHz);
– 4 MB Flash, 400 KB SRAM (8 KB RTC);
– Wi-Fi support (2.4 GHz, 11b/g/n generation, 150 Mbps, WPA3);
– Bluetooth Low Energy 5 support;
– Crypto Accelerators (RSA, AES, HMAC);
– Interfaces (USB Serial, SPI, CAN-FD/TWAI, UART, I2C, I2S).

The chip is equipped with a 32-bit arithmetic logic unit (ALU) that supports only fixed-point calculations. The instruction set (hardware-supported) is compatible

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

with RV32IMC ISA (meaning: I - basic set of 32-bit instructions, M - integer multiplication and division, C - shortened/compressed versions of instructions that reduce code size).

4.1 Algorithm implementation

The implementation of the presented algorithms (Algo) was done in C (according to RISCV32 GCC). C provides better control over the implementation compared to MicroPython (especially in the context of low-level hardware instructions). All code was compiled using 'riscv32-esp-elf-gcc' (ver. 14.2.0) with the optimization flag '-Og' (typical configuration for run tests and analyzes). All declared variables and constants (including arrays) were declared as fixed-point (usually 32-bit, thus maintaining a "reserve" for overflows). Where multiplication or division operations occur, variables are extended to 64 bits. This allows full control over the bit range and no loss of calculation precision. After the operations are performed, the variables are reduced back to 32 bits. The algorithm can also be easily implemented on 16 bits. However, with a 32-bit unit available, reducing variables to 16 bits does not bring significant benefits as all hardware calculations are performed in 32-bit precision.

The main calculation function has been prepared for both tanh() and sigmoid() operations. It is based on the CORDIC algorithm, which is based on two tables (LUT) xx_h and yy_h and iterative operations performed in a "for" loop. Multiplication and division by 2^{20} have been implemented as multiple bit/arithmetic shifts (left for multiplication, right for division). The operation $K \cdot 20^{20}$ has been calculated as a constant at the compilation stage, thus reducing the number of operations performed.

The sigmoid function is not directly implemented in the math library. This function is implemented according to the equation:

$$sigmoid(x) = \frac{1.0}{1.0 + e^{-x}}$$

where the exponential function is directly used from the math library. The equation was implemented using double-precision floating-point operations to maintain the highest accuracy for the reference values.

4.2 Measurement methodology

For modern central processing units (CPUs), analyses frequently focus on performance, electrical power usage and overall energy efficiency, particularly in the context of high-performance computing systems. In contrast, graphics processing units (GPUs) are often evaluated using similar criteria, with attention given to how effectively they execute large numbers of parallel operations. Performance in both cases is typically defined by the time required to complete specific computational tasks at a given accuracy level and workload intensity, while accounting for resource utilization such as bus bandwidth, cache behavior,

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

and processor cycle consumption. Numerous recent studies in the literature rely on these metrics to present and compare experimental results [5,1,12,4,6].

In our scenario, the following parameters are analyzed to verify the operation of computational algorithms implemented in microprocessor systems:

- error bounds (the basic factor determining the correct operation of the algorithms);
- execution speed (a factor determining the optimal computational complexity that allows competition with library functions);
- program memory usage (a factor determining the optimal implementation using hardware commands rather than program instructions).

The following measurements were made for $2^{17} = 131072$ samples. This is a complete overview of the input data in the declared Q1.16 format (17 bits in total). The individual measurements were implemented as follows:

- error measurements: the presented fixed-point algorithm (Algo) and SoftFloat library functions (Lib) were implemented, and the results from Lib dumped to the same fixed-point precision (16 bits after the decimal point). The results obtained in this way were compared to the values obtained from library functions in double precision floating point. Errors (differences from the obtained values) were determined and statistical values were calculated.
- speed measurements: it was based on readings from a counter (timer) clocked at 40 MHz (this is the maximum possible speed for both Algo and Lib; at higher speeds, the timer did not start). Before executing a given calculation function tanh() or sigmoid(), the timer is reset, and after execution, the state of the timer register is recorded. The values obtained were summed for successive samples and converted to the appropriate time units.
- program occupancy measurement: the presented Algo and Lib functions were implemented. The remaining code was reduced to the minimum necessary (only calculations for Algo and Lib, without support for microcontroller peripheral elements, e.g. communication). The occupancy values were read from the information after the project compilation.

The tests required the preparation of three separate implementation projects (three isolated experiments). This allowed for the separation of measurements and guaranteed that other functions would not affect key operations. To measure time, the total calculation time for 2^{17} samples and the average time for one sample were determined. To measure occupancy, an additional verification of the occupancy of "clean" code was performed, i.e., only the necessary startup configuration with empty loops in the code. This made it possible to determine the occupancy of the algorithms themselves (without additional necessary startup code). Reference values used for error computation were obtained from the standard C 'math.h' library evaluated in 64-bit double-precision floating-point arithmetic. When measuring errors, the following statistical values were determined:

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

– smallest error (Min), calculated by:

$$err_{min} = MIN\{err(i)\} \text{ for } i = 1, 2, ...N,$$

– largest error (Max), calculated by:

$$err_{max} = MAX\{err(i)\} \text{ for } i = 1, 2, ...N,$$

– average error, calculated by:

$$err_{ave} = \frac{\sum_{i=1}^{N} err(i)}{N}$$

– mean absolute error, calculated by:

$$err_{abs} = \frac{\sum_{i=1}^{N} |err(i)|}{N}$$

– standard deviation, calculated by:

$$err_{std} = \sqrt{\frac{\sum_{i=1}^{N} [(err(i) - err_{ave})^2]}{N}}$$

where: $N = 2^{17}$, $err(i) = q(i) - q_{fp}(i)$, $q(i)$ - output value of the function in the selected implementation, $q_{fp}(i)$ - output value of the function in full precision.

4.3 Measurement results

All tests were performed on the ESP32-C3FH4 chip. The results of individual tests are presented in Tables 1, 2 and 3.

When analyzing the errors, it can be seen that the standard deviation of errors for the Algo was smaller than the Lib functions (when casting to fixed-point format Q1.16). This shows that the developed Algo is correct and generally has fewer errors. The minimum and maximum values indicate that there are no significant errors deviating from the accepted precision of Q1.16 (where the weight of the least significant bit (LSB) is $2^{-16} \approx 1.5259e^{-05}$). While maintaining errors at a satisfactory level (and comparable to Lib functions), execution times were significantly reduced. For the implementation of the presented tanh() algorithm, the time was reduced by over 83%, while for sigmoid() the time was reduced by over 79% - for the average time to determine the result of a single sample. The measurement of absolute occupancy is burdened by the inclusion of additional runtime code (base program). Therefore, it was important to determine the relative occupancy with respect to the base program. These values clearly indicate that the appropriate adaptation of dedicated algorithms to the processor architecture (e.g., fixed-point operations) can significantly reduce code occupancy. The reduction in memory program occupancy is almost 30% for tanh() and almost 18% for sigmoid().

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

Table 1. Computation times of individual implementations of the tanh() and sigmoid() algorithms.

Realisation	Full execution time for 2^{17} samples	Average execution time for one sample
Presented Algo of tanh()	665.32[ms]	5.0759609[us]
Function of tanh() base on Lib	3965.24[ms]	30.252378[us]
Presented Algo of sigmoid()	554.74[ms]	4.2322975[us]
Function of sigmoid() base on Lib	2798.14[ms]	21.348151[us]

Table 2. Program memory usage of individual implementations of the tanh() and sigmoid() algorithms.

Realisation	Full implementation of the program	Relative occupancy with respect to the base program
Presented Algo of tanh()	274882[B]	2222[B]
Function of tanh() base on Lib	275818[B]	3158[B]
Presented Algo of sigmoid()	274902[B]	2242[B]
Function of sigmoid() base on Lib	275384[B]	2724[B]
Base program (only necessary configuration and empty loops)	272660[B]	-

Table 3. Errors of individual implementations of the tanh() and sigmoid() algorithms.

Realisation	Min error Max Error	Mean error	Mean abs. error	Standard deviation
Presented Algo of tanh()	-1.31077e-05 1.42400e-05	-3.69464e-07	3.69464e-07	2.62434e-06
Function of tanh() base on Lib	-1.52513e-05 7.90624e-06	-7.56369e-06	7.56369e-06	4.40971e-06
Presented Algo of sigmoid()	-9.45004e-06 7.83283e-06	-3.57385e-07	3.57385e-07	1.97767e-06
Function of sigmoid() base on Lib	-1.52588e-05 -1.12144e-11	-7.63221e-06	7.63221e-06	4.40192e-06

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

5 Conclusion

In this work, we introduced a unified, integer-only activation kernel for `tanh(x)` and `sigmoid(x)`, specifically designed for RISC-V microcontrollers without hardware floating-point support. By combining LUT anchoring, a compact sequence of hyperbolic CORDIC microrotations, and a final linear post-rotation in Q20 arithmetic, the proposed method achieves deterministic accuracy better than one ULP in Q1.16 across the entire 17-bit input domain. The unified structure eliminates the need for separate implementations of hyperbolic functions, reducing both software and hardware complexity.

Experimental evaluation on an ESP32-C3 RISC-V microcontroller demonstrates that the kernel provides substantial performance benefits, with more than an 80% reduction in execution time compared to SoftFloat-based library functions, while simultaneously lowering program memory usage. Importantly, the error characteristics remain competitive with—and in many cases superior to—standard math-library implementations when cast to fixed-point formats. These results confirm that carefully designed fixed-point algorithms can deliver high-accuracy nonlinear activation functions at a fraction of the computational cost, making them highly suitable for embedded AI workloads.

Beyond the immediate performance gains, the unified-kernel approach also simplifies integration into larger inference pipelines, enabling more predictable timing behavior and easier hardware mapping. Its deterministic structure makes it suitable for lightweight hardware extensions, such as custom RISC-V instructions or FPGA mapping, which could further reduce latency. The present implementation still relies on an integer division in the final scalar post-processing stage and has been evaluated only for `tanh` and `sigmoid` over the 17-bit internal recoded domain implemented by the kernel. Although the experimental platform is RISC-V, the computational scheme itself is not tied to RISC-V-specific instructions and can be transferred to other 32-bit processors without hardware floating-point support. Future work will extend the method to additional nonlinearities, evaluate its behavior over conventional application-level input ranges such as $[-8, 8]$, and investigate custom RISC-V instructions as well as FPGA-based mappings.

Acknowledgments. This research was partly supported by PLGrid Infrastructure at ACK Cyfronet AGH, Krakow, Poland. This work was also partly supported by the National Science Foundation under grant #2331153.

References

1. Ciznicki, M., Kopta, P., Kulczewski, M., Kurowski, K., Gepner, P.: Elliptic solver performance evaluation on modern hardware architectures. Parallel Processing and Applied Mathematics **8384** (2014). `https://doi.org/10.1007/978-3-642-55224-3_16`
2. Dequino, A., Bompani, L., Benini, L., Conti, F.: Optimizing bfloat16 deployment of tiny transformers on ultra-low power extreme edge socs. J. Low Power Electron. Appl. **15**(1), 8 (2025), `https://www.mdpi.com/2079-9268/15/1/8`

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47

3. Foundation, R.V.: Bfloat16 extension specification for risc-v. Online specification (2023), https://docs.riscv.org/reference/isa/unpriv/bfloat16.html
4. Gepner, P.: Using avx2 instruction set to increase performance of high performance computing code. Computing and Informatics **36**(5), 1001–1018 (2017)
5. Gepner, P., Fraser, D.L., Kowalik, M.F.: Second generation quad-core intel xeon processors bring 45 nm technology and a new level of performance to hpc applications. Computational Science – ICCS 2008, ICCS 2008 **5101** (2008)
6. Gepner, P., Gamayunov, V., Fraser, D.L.: Effective implementation of dgemm on modern multicore cpu. Procedia Computer Science **9**, 126–135 (2012). https://doi.org/10.1016/j.procs.2012.04.014
7. Hauser, J.R., Dunkels, A.: Softfloat source documentation. Online documentation, DBT-RISE Project (2020), https://git.minres.com/DBT-RISE/DBT-RISE-TGC/src/commit/fe3ed495199bc5ccb7bcc8cd38f848000af48c99/softfloat/doc/SoftFloat-source.html
8. Hingu, C., Fu, X., Saliyu, T., Hu, R., Mishan, R.: Power-optimized field-programmable gate array implementation of neural activation functions using continued fractions for ai/ml workloads (2024), https://doi.org/10.3390/electronics13245026
9. Hoyer, I., Utz, A., Lüdecke, A., Kappert, H., Rohr, M., Antink, C.H., Seidl, K.: Design of hardware accelerators for optimized and quantized neural networks to detect atrial fibrillation in patch ecg device with risc-v. Sensors (2023)
10. Intel: oneAPI Deep Neural Network Library (oneDNN) Developer Guide and Reference (2025), https://www.intel.com/content/www/us/en/docs/onednn/developer-guide-reference/2025-2/bfloat16-training.html
11. Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D.T., Jammalamadaka, N., Huang, J., Yuen, H., Yang, J., Park, J., Heinecke, A., Georganas, E., Srinivasan, S., Kundu, A., Smelyanskiy, M., Kaul, B., Dubey, P.: A study of bfloat16 for deep learning training. arXiv preprint arXiv:1905.12322 (2019), https://arxiv.org/abs/1905.12322
12. Kopta, P., Kulczewski, M., Kurowski, K., Piontek, T., Gepner, P., Puchalski, M., Komasa, J.: Parallel application benchmarks and performance evaluation of the intel xeon 7500 family processors. Procedia Computer Science **4**, 372–381 (2011). https://doi.org/10.1016/j.procs.2011.04.039
13. Kundu, A., Heinecke, A., Kalamkar, D., Srinivasan, S., Qin, E.C., Mellempudi, N.K., Das, D., Banerjee, K., Kaul, B., Dubey, P.: K-tanh: Efficient tanh for deep learning. arXiv preprint arXiv:1908.11263 (2019), https://arxiv.org/abs/1908.11263
14. Li, L., Gautschi, M., Benini, L.: Approximate div and sqrt instructions for risc-v: An efficiency vs. accuracy analysis. International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS) (2017)
15. Li, Z., Zhang, Y., Sui, B., Xing, Z., Wang, Q.: Fpga implementation for the sigmoid with piecewise linear fitting method based on curvature analysis (2022), https://doi.org/10.3390/electronics11091365
16. Rizwan, R., Noor, M., Rehman, Z., Imran, U.: Accelerating ai on risc-v: Optimizing bfloat16 for improved efficiency. RISC-V Summit Europe, Munich, 2024 (2024)
17. Tak, P., Tang, P.: An open-source risc-v vector math library. 2024 IEEE 31st Symposium on Computer Arithmetic (ARITH) (2024)
18. Yıldız, R.O., Yilmazer-Metin, A.: Cordic accelerator for risc-v. 29th Telecommunications forum TELFOR 2021 (2021)
19. Zamirai, P., Zhang, J., Aberger, C.R., Sa, C.D.: Revisiting bfloat16 training. arXiv preprint arXiv:2010.06192 (2021), https://arxiv.org/abs/2010.06192

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_47