Fast Trigonometric Functions using the RLIBM Approach

Sehyeok Park
Rutgers University
New Brunswick, NJ, USA
sp2044@cs.rutgers.edu

Santosh Nagarakatte
Rutgers University
New Brunswick, NJ, USA
sn349@cs.rutgers.edu

This paper describes our experience developing polynomial approximations for trigonometric functions that produce correctly rounded results for multiple representations and rounding modes using the RLIBM approach. A key challenge with trigonometric functions concerns range reduction with π, which reduces a given input in the domain of a 32-bit float to a small domain. Any rounding error in the value of π is amplified during range reduction, which can result in wrong results. We describe our experience implementing fast range reduction techniques that maintain a large number of bits of π both with floating-point and integer computations. The resulting implementations for trigonometric functions are fast and produce correctly rounded results for all inputs for multiple representations up to 32-bits with a single implementation.

1 Introduction

Scientific computing extensively uses elementary functions (*e.g.*, e^x) provided by math libraries. Producing correctly rounded results for all inputs for elementary functions is challenging (*i.e.*, table-maker's dilemma [14]). Hence, mainstream math libraries do not produce correct results for all inputs. The lack of correctly rounded math libraries can make the application non-portable and can cause reproducibility issues. An application can produce totally different results on two different machines.

A common approach to develop math libraries is to generate minimax polynomial approximations [7, 20], which minimize the maximum error across all inputs with respect to the real value. Eventually, the real coefficients of the generated polynomial are rounded to a hardware supported FP representation [6, 5]. By using polynomial approximations with sufficiently large degrees, such minimax approaches can generate correctly rounded elementary functions for a single representation [9].

As an alternative, our RLIBM project [19, 17, 18, 1] makes a case for approximating the correctly rounded result directly rather than the real value of an elementary function. The RLIBM project uses the MPFR high-precision math library [12] as the oracle and focuses on generating efficient implementations given the oracle result. Once the oracle correctly rounded result for a given input is known, the main insight in the RLIBM project is that there is an interval of real values around the correctly rounded result such that producing any real value in this interval rounds to the correctly rounded result. With the RLIBM approach, the freedom available to the polynomial generation step is larger than the freedom available with minimax methods (*i.e.*, it is 1 units in the last place, ULP, for all inputs). In contrast, minimax methods have significantly smaller freedom when the real-valued output of an elementary function is very close to the rounding boundary (*e.g.*, midpoint of two FP values with round to nearest modes) [15, 30]. Using the interval around the correctly rounded result, the RLIBM project structures the task of generating a polynomial of degree d that produces correctly rounded results for all inputs as a linear program (LP) (*i.e.*, a system of linear inequalities). It uses an LP solver to identify the coefficients [1, 2].

With the increased use of custom formats [23, 29], the RLIBM project proposes a novel idea to generate a single polynomial approximation that produces correctly rounded results for multiple representations and rounding modes [19], which can serve as a reference library. To generate correctly

Submitted to:
VSS 2025

© S. Park & S. Nagarakatte
This work is licensed under the
Creative Commons Attribution License.

rounded results for FP representations with up to n-bits that have E-bits for the exponent, the key insight is to generate a polynomial approximation that produces correctly rounded results for a representation with $(n+2)$-bits with the *round-to-odd* rounding mode [19]. When such a result is double rounded to the target representation, it produces correct results for representations with k bits, where $E+2 \leq k \leq n$, and for all standard rounding modes.

We have been trying to generate fast correctly rounded trigonometric functions with the RLIBM approach for a few years. We were not successful earlier because we did not know how to perform both correct and efficient range reduction for these functions. The excellent reference on elementary functions by Muller [21] describes algorithms for range reduction for trigonometric functions. However, naïvely implementing these algorithms can slow down resulting implementations by 2-3$\times$ when compared to final fast implementations that we describe in this paper. Further, it is necessary to compute with a large number of digits of π efficiently. When we maintained an insufficient number of bits of π, numerical errors in range reduction performed with double precision often resulted in conflicting constraints in the linear program generated with the RLIBM approach.

The first step for generating any correctly rounded elementary function is range reduction, which reduces an input in the domain of floating-point (FP) to a small domain where a polynomial approximation is feasible. A key challenge with range reduction for trigonometric functions is that one would need to maintain a large number of bits of $\frac{1}{\pi}$. Given an input x, one can perform range reduction as $x' = |x| - k\pi$ to produce the reduced input $x' \in [0, \pi)$, where $k = \lfloor \frac{|x|}{\pi} \rfloor$. Alternatively, one can also perform a symmetric range reduction $x - k\pi$ using $k = \lceil \frac{x}{\pi} \rfloor$, where $\lceil \frac{x}{\pi} \rfloor$ computes the nearest integer when $\frac{x}{\pi}$ is rounded, to obtain $x' \in [-\frac{\pi}{2}, \frac{\pi}{2}]$. If such range reduction is applied using just 64-bits of $\frac{1}{\pi}$, the reduced input will have no accurate bits when x is a large number.

This paper describes our experience exploring efficient range reduction techniques for trigonometric functions, which is performed using a combination of FP and integer operations, for generating a single polynomial approximation that produces correct results for all inputs with multiple representations and rounding modes. Although the core algorithms for range reduction of trigonometric functions are well-known [24, 21, 22, 11, 28, 9, 10, 26, 16, 8], the challenge is in efficiently implementing them with sufficient accuracy to produce correctly rounded results for all inputs. We observe that efficient range reduction is as important or more important than low-degree polynomial approximations for trigonometric functions. We describe the algorithms and evaluate the performance implications of various range reduction techniques. We believe this exposition will help future implementers of math libraries. Using these range reduction techniques, we have implemented correctly rounded trigonometric functions where a single implementation produces correctly rounded results for multiple representations from 10-bits to 32-bits for all the standard rounding modes. It is faster or similar in performance compared to mainstream libraries and other correctly rounded libraries. Our integer-based range reduction improves the performance by 19% when compared to our FP-based strategy. *In summary, the efficient range reduction methods described in this paper enabled us to generate correct and fast trigonometric functions after multiple years of trying to use the* RLIBM *approach for these functions.*

2 Background on the RLIBM Approach

The RLIBM approach [17, 18, 19, 1] assumes the existence of an oracle that provides correctly rounded results (*e.g.*, the MPFR library) and focuses on generating efficient implementations. Given the oracle result, the RLIBM project makes a case for approximating the correctly rounded result rather than the real value of an elementary function. Typically, the implementation of a correctly rounded function for

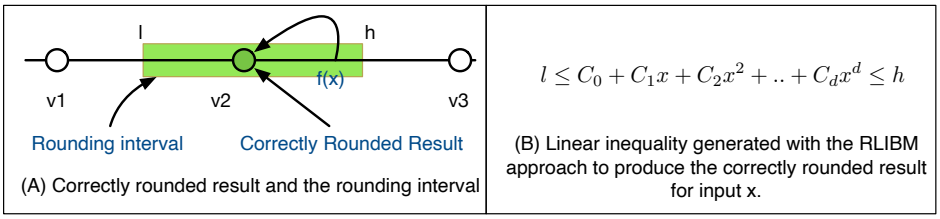

Figure 1: (A) The rounding interval of a correctly rounded result v2. (B) The linear inequality generated with the RLIBM approach to produce the correctly rounded result for an input x given the rounding interval $[l,h]$.

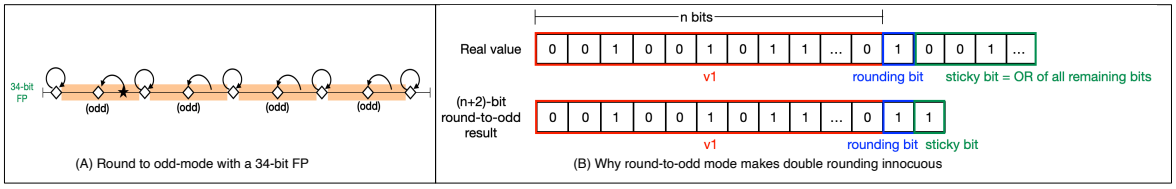

Figure 2: (A) Illustration of round-to-odd rounding mode with a 34-bit FP representation. (B) Intuition on why double rounding is innocuous with the round-to-odd mode. Rounding the round-to-odd result to a target representation has the same truncated value, rounding bit, and sticky bit as rounding that real value directly to the given target representation.

a 32-bit float representation uses the 64-bit double precision representation internally. After a correctly rounded result for a 32-bit float input is available using an oracle, the RLIBM project observes that there is an interval of values in the double precision representation around the correctly rounded result such that any value in that interval rounds to the correctly rounded value (see Figure 1(A)). This interval is called the rounding interval. The rounding interval is represented as $[l,h]$, where l is the lower bound and h is the upper bound. When the goal is to generate a polynomial of degree d with $d+1$ terms, the rounding interval implies a linear constraint on the result of the polynomial evaluation for a given input x as shown in Figure 1(B). A 32-bit representation has 2^{32} inputs and the RLIBM project generates a system of linear inequalities corresponding to the 32-bit inputs and their respective rounding intervals. Now, the task of generating a correctly rounded function boils down to the task of identifying the coefficients (*i.e.*, C_i's) of a polynomial that satisfies these inequalities. Hence, the RLIBM project frames the problem of generating correct elementary functions as a linear program and uses an LP solver to solve it.

Range reduction techniques that reduce the original input from the domain of a 32-bit float representation (*i.e.*, $[2^{-149}, 2^{128})$) to a small domain (*e.g.*, $[-1,1]$) are crucial to generate approximations. The original input x is range reduced to x'. The polynomial approximation computes the result for x'. The result is output compensated to produce the final output for x. Both range reduction and output compensation are performed in the double precision representation and can have numerical errors. To account for the numerical errors, the RLIBM project deduces intervals for the reduced domain such that the polynomial evaluation over the reduced input produces correct results for the original inputs. The RLIBM project uses the inverse of the output compensation function to infer the reduced rounding intervals. Finally, a system of linear inequalities corresponding to the reduced inputs and the reduced rounding intervals are solved using an LP solver iteratively to identify coefficients of a polynomial of degree d.

Multiple representations and rounding modes. The RLIBM approach described till now produces correctly rounded results for all inputs for a single representation and a single rounding mode. There are

four rounding modes in the standard. Further, many new representations that vary either the dynamic range or the precision are being proposed (*e.g.*, bfloat16, tensorfloat32, FP8). A single implementation that provides correct results for these representations is appealing as a reference library. One approach to generate a single approximation for multiple representations is to use a correctly rounded function designed for a higher precision representation. Then, round the result to the target representation. However, it produces wrong results for some inputs because of double rounding errors. The first rounding happens when the real value is rounded to produce the correctly rounded result of a higher precision representation. The second rounding happens when that result is rounded to the target representation.

The RLIBM project has proposed a method to generate a single polynomial approximation that produces correctly rounded results for all FP representations up to n-bits [19]. The key idea is to generate a polynomial for the (n+2)-bit representation (which has 2 additional precision bits compared to the n-bit representation) using the *round-to-odd* rounding mode. The round-to-odd result with two additional precision bits retains all necessary information to produce the correct results for any representation up to n-bits. In the round-to-odd mode [4, 13], if a real value is exactly representable by the target representation, we represent it with that value. Otherwise, the value is rounded to an adjacent floating point value whose bit-pattern is odd when interpreted as an integer. Figure 2(A) illustrates the round-to-odd mode. To understand why double rounding with the round-to-odd mode produces correct results, we have to first understand how rounding works. Typically, we need three pieces of information when rounding a real value to an n-bit FP representation: (1) the first n-bits of the real value in the binary representation, (2) the $(n+1)^{th}$-bit known as the rounding bit, and (3) the result of the bitwise OR of all the remaining bits, known as the sticky bit. When we round a real value to a (n+2)-bit representation using the round-to-odd mode, the round-to-odd result precisely maintains the same three pieces of information as rounding the real value directly to the target representation of a lower bitwidth as shown in Figure 2(B). Hence, subsequent rounding of the round-to-odd result to a representation with less than or equal to n-bits produces the correct result. This ability to produce correctly rounded results for multiple representations and rounding modes with a single polynomial approximation is feasible because the RLIBM project directly approximates the correctly rounded result rather than the real value.

3 Fast Range Reduction with Sufficient Accuracy

We describe range reduction techniques and implementation choices that finally enabled us to build fast and correct trigonometric functions with the RLIBM approach. During range reduction, the original input x in the domain of a 32-bit float is mapped to a reduced input x'. This process involves π (*i.e.*, $x' = x - k\pi/2^t$). For large values of x, we need to maintain a large number of bits of $\frac{2^t}{\pi}$. We then need to perform computation with them efficiently to identify $k = [\frac{2^t x}{\pi}]$ and the reduced input. If we just maintain 64 precision bits, then the reduced input is either 0 (for large inputs) or significantly wrong (for other inputs). This range reduction is extremely challenging when the goal is to produce correctly rounded results for all inputs while providing good performance! Existing libraries make trade-offs either with performance (*e.g.*, Payne-Hanek implementations [24]) or correctness (*e.g.*, GCC's libm). Trigonometric functions are periodic functions with a period of 2π. Hence, the range reduction produces a reduced input $x' \in [-\frac{\pi}{2^{t+1}}, \frac{\pi}{2^{t+1}}]$ given the original input $x \in [-\infty, \infty]$ using periodicity, where t is an integer greater than or equal to 0. Effectively,

$$x = x' + k\pi/2^t$$

To perform the above range reduction, the key task involves computing an integer $k = [\frac{x}{\frac{\pi}{2^t}}] = [\frac{2^t x}{\pi}]$. Using k, one can compute a reduced input in $[-\frac{\pi}{2^{t+1}}, \frac{\pi}{2^{t+1}}]$ as shown below.

$$r = \frac{2^t x}{\pi} - k, \quad x' = x - k\frac{\pi}{2^t} = \frac{\pi}{2^t}\left(\frac{2^t x}{\pi} - k\right) = \frac{\pi}{2^t} r \tag{1}$$

These identities hold when the computation is performed with real numbers. For this range reduction, we need to perform computations involving $\frac{2^t}{\pi}$. Doing so requires maintaining a large number of bits of $\frac{2^t}{\pi}$. Without maintaining sufficient digits of $\frac{2^t}{\pi}$, the resulting reduced input will have very few accurate bits. This is because most of the leading bits will be canceled when x is close to a multiple of $\frac{\pi}{2^t}$ and the remaining bits will be influenced by rounding. This problem is extremely severe when x is a large number. If we maintain $\frac{256}{\pi}$ using only 64 bits, assuming $t = 8$, then the reduced input will be 0 for all inputs when $2^{80} \leq x$. This is because all the 64 precision bits of $\frac{256}{\pi}$ would contribute to k, leaving no bits available for the fractional piece $r = \frac{256x}{\pi} - k$.

Range reduction for sin(x). As described above, the main objective of range reduction for $sin(x)$ is to transform a given input $x \in [-\infty, \infty]$ into a reduced input $x' \in [-\frac{\pi}{2^{t+1}}, \frac{\pi}{2^{t+1}}]$. Leveraging the formula $x = k\frac{\pi}{2^t} + x'$ and the trigonometric property $sin(a+b) = sin(a)cos(b) + cos(a)sin(b)$, we can compute $sin(x)$ as follows.

$$sin(x) = sin\left(\frac{k\pi}{2^t} + x'\right) = sin\left(\frac{k\pi}{2^t}\right) cos(x') + cos\left(\frac{k\pi}{2^t}\right) sin(x') \tag{2}$$

Equation 2 reduces the task of generating a polynomial approximation of $sin(x)$ for an input $x \in [-\infty, \infty]$ to generating approximations of $sin(x')$ and $cos(x')$ for the reduced input $x' \in [-\frac{\pi}{2^{t+1}}, \frac{\pi}{2^{t+1}}]$. Once we choose a value for t (*e.g.*, $t = 8$ and we generate a reduced input in $[-\frac{\pi}{512}, \frac{\pi}{512}]$), the output compensation formula in Equation 2 requires us to precompute the tables for $sin(\frac{k\pi}{2^t})$ and $cos(\frac{k\pi}{2^t})$ for all possible values of k. One issue in precomputing these tables is that the set of possible values of k can be very large. We leverage the periodicity and symmetry of $sin(x)$ and $cos(x)$ to reduce the number of necessary precomputed values.

Specifically, $sin(x) = sin(x - 2m\pi)$ and $cos(x) = cos(x - 2m\pi)$ for any integer m. Applying this property to $sin(\frac{k\pi}{2^t})$ and $cos(\frac{k\pi}{2^t})$ in Equation 2 leads to a significant reduction in table sizes.

$$m = \lfloor \frac{k}{2^{t+1}} \rfloor, \;\; k' = k - 2^{t+1}m \tag{3}$$

$$sin\left(\frac{k\pi}{2^t}\right) = sin\left(\frac{k\pi}{2^t} - 2m\pi\right) = sin\left(\frac{k - 2^{t+1}m}{2^t}\pi\right) = sin\left(\frac{k'\pi}{2^t}\right) \tag{4}$$

$$cos\left(\frac{k\pi}{2^t}\right) = cos\left(\frac{k\pi}{2^t} - 2m\pi\right) = cos\left(\frac{k - 2^{t+1}m}{2^t}\pi\right) = cos\left(\frac{k'\pi}{2^t}\right) \tag{5}$$

Applying Equation 4 and Equation 5 to Equation 2, we get

$$sin(x) = sin\left(\frac{k'\pi}{2^t}\right) cos(x') + cos\left(\frac{k'\pi}{2^t}\right) sin(x') \tag{6}$$

A total of 2^{t+2} precomputed values are necessary at this point (2^{t+1} each for $sin(\frac{k'\pi}{2^t})$ for $cos(\frac{k'\pi}{2^t})$), since $k' \in [0, 2^{t+1})$. If $t = 8$, then would need a total of 1024 precomputed values. We further reduce the number of required table entries to 512 by exploiting the property that $cos(\frac{k'\pi}{2^t}) = sin(\frac{k'\pi}{2^t} + \frac{\pi}{2}) = sin(\frac{k'+2^{t-1}}{2^t}\pi)$. Using the same property as before, one can conclude that $sin(\frac{k'+2^{t-1}}{2^t}\pi) = sin(\frac{k'+2^{t-1}-2^{t+1}n}{2^t}\pi)$ where $n = \lfloor \frac{k+2^{t-1}}{2^{t+1}} \rfloor$. Since $k' + 2^{t-1} - 2^{t+1}n$ is in the range of k', the table for $sin(\frac{k'\pi}{2^t})$ contains all the possible values of $cos(\frac{k'\pi}{2^t})$. Algorithm 1a shows the sketch of the range reduction steps.

Range Reduction and Output Compensation for cos(x) and tan(x). The range reduction for $cos(x)$ is the same as that for $sin(x)$ in that it reduces an input $x \in [-\infty, \infty]$ to a reduced input $x' \in [-\frac{\pi}{2^{t+1}}, \frac{\pi}{2^{t+1}}]$ while minimizing the number of precomputed values needed for output compensation.

The output compensation formula for $cos(x)$ using k and x' is as follows.

$$cos(x) = cos\left(\frac{k\pi}{2^t} + x'\right) = cos\left(\frac{k\pi}{2^t}\right) cos(x') - sin\left(\frac{k\pi}{2^t}\right) sin(x') \tag{7}$$

Using the definitions of m and k' defined in Equation 3 and the periodicity $cos(x) = cos(x - 2m\pi)$ for any integral value of m, we can rewrite Equation 7 as follows.

$$cos(x) = cos\left(\frac{k'\pi}{2^t}\right) cos(x') - sin\left(\frac{k'\pi}{2^t}\right) sin(x') \tag{8}$$

Since $k' \in [0, 2^{t+1})$, we require a precomputed table of 2^{t+2} values in total (2^{t+1} for $sin(\frac{k'\pi}{2^t})$ and 2^{t+1} for $cos(\frac{k'\pi}{2^t})$). Similar to sin, we further reduce the size of the precomputed table to 2^{t+1} entries by leveraging the property $cos(x) = sin(x + \frac{\pi}{2})$. As the range of k' is the same for both $sin(x)$ and $cos(x)$, we use a single table containing all the possible values of $sin(\frac{k'\pi}{2})$ for the output compensation of both functions. In summary, our range reduction strategy for $cos(x)$ computes for each x an appropriate k' and x', which reduces the original problem to approximating $sin(x')$ and $cos(x')$ for $x' \in [-\frac{\pi}{2^{t+1}}, \frac{\pi}{2^{t+1}}]$. Given that $tan(x) = \frac{sin(x)}{cos(x)}$, we can also implement $tan(x)$ using the range reduction and output computation strategies described thus far and the precomputed values used for $sin(x)$ and $cos(x)$.

3.1 Efficient Range Reduction with FP Operations

The range reduction described above requires computing $k = [\frac{256x}{\pi}]$ and $x' = \frac{\pi}{256}(\frac{256x}{\pi} - k) = \frac{\pi}{256}r$ accurately. For the inputs with relatively small absolute values (*i.e.*, $|x| \leq \frac{\pi}{128}$), direct polynomial approximation is possible and thus range reduction is not necessary. Range reduction, however, is required for the remaining inputs to obtain an efficient polynomial approximation. Implementing Algorithm 1a for these inputs requires computing $\frac{256x}{\pi}$ with low latency while maintaining a large number of bits of $\frac{256}{\pi}$ for accuracy. The number of bits required for $\frac{256}{\pi}$ depends on the precision of the target representation, the exponent of the input x, and the number of accurate bits needed in the reduced input x'. Worst case analysis [24, 21, 22] for range reduction suggests that approximately 200-bits of $\frac{256}{\pi}$ are sufficient for performing range reduction with high accuracy for all 32-bit inputs.

Algorithm 1b, which is based on LLVM's range reduction [27], provides an implementation of Algorithm 1a for small inputs. For relatively small inputs (*i.e.* $|x| < 2^{30}$), we observe that approximately 80-bits of $\frac{256}{\pi}$ spread across two double-precision values suffice for computing a reduced input with desirable accuracy. The double 0x*1.45f306c*p+6 (line 2) represents the first 28-bits of $\frac{256}{\pi}$. The value

```
1  Function range_reduction(x)
2      k = [256x/π];
3      r = 256x/π − k;
4      x' = π/256 r ;
5      m = ⌊k/512⌋;
6      k' = k − 512m;
7      return (x', k')
8  end
```

(a)

```
1  Function fp_range_reduction_small(x)
2      p0 = 0x1.45f306cp+6∗x;
3      p1 = 0x1.c9c882a53f84fp-22∗x;
4      p = p0 + p1;
5      p_int = round(p);
6      k = (int64_t)p_int;
7      r = (p0 − p_int) + p1;
8      xp = r ∗ PI_OVER_256;
9      kp = k & 0x1ff;
10     return (xp, kp);
11 end
```

(b)

Algorithm 1: (a) High level range reduction producing a reduced input $x' \in [-\frac{\pi}{512}, \frac{\pi}{512}]$ and the index k' for pre-computed tables. Here, $[\frac{256x}{\pi}]$ computes the integer nearest to $\frac{256x}{\pi}$. (b) A FP-based implementation of Algorithm 1a that produces reduced inputs for small inputs (*i.e.*, $|x| < 2^{30}$). The values xp and kp denote x' and k' respectively.

0x*1.c9c882a53f84f*p-22 represents the subsequent 53-bits of $\frac{256}{\pi}$ obtained via round-to-nearest. Because 0x*1.45f306c*p+6 contains only 28-bits of precision, the product $p0$ =0x*1.45f306cp*+6$*x$ has at most 52 precision bits and is thus exactly representable as a double value. By avoiding intermediate rounding errors, Algorithm 1b can compute the integer k and an accurate approximation of the fractional value $r = \frac{256x}{\pi} - k$ using the partial products $p0$ =0x*1.45f306c*p+6$*x$ and $p1$ = 0x*1.c9c882a53f84f*p-22$*x$.

The algorithm in Figure 5(a) sketches an FP-based implementation of the target range reduction for larger inputs (*i.e.*, $2^{30} \leq |x|$). The algorithm's objective is to appropriately divide the bits of $\frac{256}{\pi}$ to avoid rounding (and the concomitant rounding errors) wherever possible. For large inputs, the initial bits of $\frac{256}{\pi}$ are insufficient. To account for a wider range of inputs, we generate 196-bits of $\frac{256}{\pi}$ using the MPFR library and store them as 28-bit pieces in an array of doubles (*i.e.*, _256_*over_pi*_28 in Figure 5(a)). Each element is generated using the round-to-zero mode (*i.e.*, truncation). We split $\frac{256}{\pi}$ into 28-bit pieces to ensure that no partial product (*i.e.*, _256_*over_pi*_28$[idx] * x$) incurs rounding error. The product of a 28-bit piece of $\frac{256}{\pi}$ and a 32-bit input with 24-bits of precision is exactly representable as a double.

The initial task in the algorithm in Figure 5(a) is to identify the pieces of $\frac{256}{\pi}$ necessary for computing the fractional bits of $\frac{256x}{\pi}$ and the relevant portions of k, which we store as a 64-bit integer. For a very large input, k can exceed the dynamic range of `int64_t`. However, we only require the lower 9-bits of k (see Equation 6). Hence, we can skip over the pieces of $\frac{256}{\pi}$ for which the exponent of the least significant bit exceeds 8 when summed with the exponent of the least significant digit of the input (see 7-9 in Figure 5(a)). Once the first piece that contributes to the least 9-bits of k is identified, we proceed to compute the partial products of $\frac{256x}{\pi}$. At most three partial products can contribute to the lower order bits of k (lines 11-20). After obtaining the relevant integer bits, we compute the fractional portion of $\frac{256x}{\pi}$ denoted r by accumulating the residual fractional bits and the final partial product (*i.e.* _256_*over_pi*_28$[idx+3] * x$ in line 22). Multiplying r from this algorithm with $\frac{\pi}{256}$, which is maintained in double precision, produces the reduced input x'.

The iterative search for the first relevant piece of $\frac{256}{\pi}$ (lines 7-9) is a key source of overhead in the algorithm in Figure 5(a). Another drawback is that the limited precision of each piece in _256_*over_pi*_28 requires computing the relevant bits of k across many elements (up to 3) of the array. We present an alternative strategy that addresses these issues in Figure 5(b). This alternative range reduction method splits $\frac{256}{\pi}$ into double values with 53-bits of precision stored in the array _256_*over_pi*_53. As before, we

```
1  Function fp_range_reduction_v1(x)
2      int64_t k,kp;
3      z = |x|;
4      biased_exp = bits(z) >> 23;
5      lsb_exp = biased_exp − 150;
6      idx = 0;
7      while lsb_exp + _256_over_pi_exp[idx] > 8 do
8          idx++;
9      end
10     prod_hi = (double)x ∗ _256_over_pi_28[idx];
11     k_hi = round(prod_hi);
12     k_hi_int = (int64_t)k_hi;
13     k_hi_int = k_hi_int & 0x1ff;
14     frac_hi = prod_hi − k_hi;
15     prod_mid = (double)x ∗ _256_over_pi_28[idx+1] + frac_hi;
16     k_mid = round(prod_mid);
17     k_mid_int = (int64_t)k_mid;
18     frac_mid = prod_mid − k_mid;
19     prod_lo = (double)x ∗ _256_over_pi_28[idx+2] + frac_mid;
20     k_lo = round(prod_lo);
21     frac_lo = prod_lo − k_lo;
22     r = (double)x ∗ _256_over_pi_28[idx+3] + frac_lo;
23     xp = r ∗ PI_OVER_256;
24     k = (int64_t)(k_lo + k_mid) + k_hi_int;
25     kp = k & 0x1ff;
26     return (xp,kp);
27 end
```

(a)

```
1  Function fp_range_reduction_v2(x)
2      int64_t k,kp;
3      z = |x|;
4      biased_exp = bits(z) >> 23;
5      lsb_exp = biased_exp − 150;
6      idx = 55 <= lsb_exp;
7      prod_hi = (double)x ∗ _256_over_pi_53[idx];
8      k_hi = round(prod_hi);
9      k_hi_int = (int64_t)k_hi;
10     k_hi_int = k_hi_int & 0x1ff;
11     frac_hi = fma(x, _256_over_pi_53[idx], −k_hi);
12     prod_lo = fma(x, _256_over_pi_53[idx+1], frac_hi);
13     k_lo = round(prod_lo);
14     k_lo_int = (int64_t)k_lo;
15     r = fma(x, _256_over_pi_53[idx+1], frac_hi − k_lo);
16     r = fma(x, _256_over_pi_53[idx+2], r);
17     xp = r ∗ PI_OVER_256;
18     k = k_hi_int + k_lo_int;
19     kp = k & 0x1ff;
20     return (xp,kp);
21 end
```

(b)

Figure 5: (a) The FP-based implementation of Algorithm 1a for large inputs. We denote the reduced input and the lower 9-bits of k as xp and kp, respectively. We store 7 28-bit pieces of $\frac{256}{\pi}$ in a double array named _256_*over_pi*_28. We maintain the exponent of the last bit of each piece in the array _256_*over_pi*_28_*exp*. Here, k and *k_hi_int* are signed 64-bit integers, *idx* is a signed 32-bit integer, and the rest are FP double values. The function $bits(z)$ provides the IEEE-754 FP bit-pattern of the input as an integer. (b) An alternative FP-based strategy for large inputs. We store 4 53-bit pieces of $\frac{256}{\pi}$ in a double array named _256_*over_pi*_53. The fused-multiply-add (fma) operation that performs both FP multiplication and addition with a single instance of rounding.

generate each element of _256_*over_pi*_53 using the MPFR library, albeit in the round-to-nearest mode. The increased bit-width confines the first piece contributing to the relevant bits for k to the first two elements. The largest possible value of *lsb_exp*, which represents the exponent of the least significant bit of the input, is $254-150=104$ given that the largest possible unbiased exponent for a non-infinity, non-*NaN* 32-bit float is 254. The exponent of the least significant bit of _256_*over_pi*_53$[1]$ is -99, and thus the largest possible exponent for the least significant bit of the product _256_*over_pi*_53$[1]*x$ is $104-99=5$. Therefore, either the first or second element of _256_*over_pi*_53 is necessary to compute the 9 lowest bits of k for all inputs. Whether the first element is needed depends on the value of *lsb_exp*. The exponent of the least significant bit of _256_*over_pi*_53$[0]$ is -46. For all inputs x such that *lsb_exp* ≥ 55, the exponent of the least significant bit of _256_*over_pi*_53$[0]*x$ evaluated in infinite precision is at least $-46+55=9$. As such, the first piece of _256_*over_pi*_53 does not contribute to the 9 lowest bits k for any input such that *lsb_exp* ≥ 55. The comparison $55 \leq$ *lsb_exp* (line 6) thus determines the first relevant piece of _256_*over_pi*_53. By limiting the starting point to the first two elements of _256_*over_pi*_53, the algorithm in Figure 5(b) avoids the iterative search in Figure 5(a).

Unlike the algorithm in Figure 5(a), which divides $\frac{256}{\pi}$ into 28-bit pieces to avoid rounding during multiplication, the algorithm in Figure 5(b) is susceptible to more intermediate rounding errors as the

product of x and an element of _256_over_pi_53 may not be exactly representable as a double. To minimize this rounding error, we use fused-multiply-add instructions (lines 11-16 in Figure 5(b)), which perform multiplication and addition sequentially with a single instance of rounding [3]. By computing the partial products (*i.e.*, $_256_over_pi_53[idx] * x$) while subtracting portions of the previous products that contribute to integer bits through `fma` instructions (see lines 11 and 15), we are able to compute the fractional bits composing r (line 16) with sufficient precision in Figure 5(b).

3.2 Efficient Range Reduction with Integer Operations

After we implemented the above range reduction strategy with FP operations, we observed that the resulting implementations produced correctly rounded results but were slower than mainstream libraries. Hence, we subsequently explored integer-based implementations that compute $\frac{256x}{\pi}$. Moreover, we can compute with more precision than FP doubles by using integers (*i.e.*, `uint64_t` and `uint128_t`), which can reduce the number of intermediate products. In the earlier FP-based strategies, maintaining the pieces of $\frac{256}{\pi}$ and the partial products as doubles helped us identify the integer k and the fractional parts for the reduced input easily. With our integer based implementation, we maintain the pieces of $\frac{256}{\pi}$ and partial products as integers and we track the exponent implicitly. Finally, we compute the reduced input and k using bitwise shift operations.

Range reduction with small inputs. For inputs such that $|x| < 2^{30}$ (excluding special case inputs that do not require range reduction), we compute k and the reduced input using the first 80-bits of $\frac{256}{\pi}$. As we want to primarily compute with integers, we represent the significand of the input x as an integer (*i.e.*, multiplying by 2^{23}, which is equivalent to subtracting 23 from the exponent of x). For these inputs, the integer k is contained in the first 64-bits of the product $\frac{256x}{\pi}$ and we can generate precise reduced inputs using 80-bits of $\frac{256}{\pi}$. We generate 80 bits of $\frac{256}{\pi}$ and store 40 bits of $\frac{256}{\pi}$ in two 64-bit numbers $P1$ and $P0$. We store 40-bits of $\frac{256}{\pi}$ each in $P1$ and $P0$ because we can store $P1 * x$ and $P0 * x$ exactly as 64-bit integers. The partial products are added as shown in Figure 6.

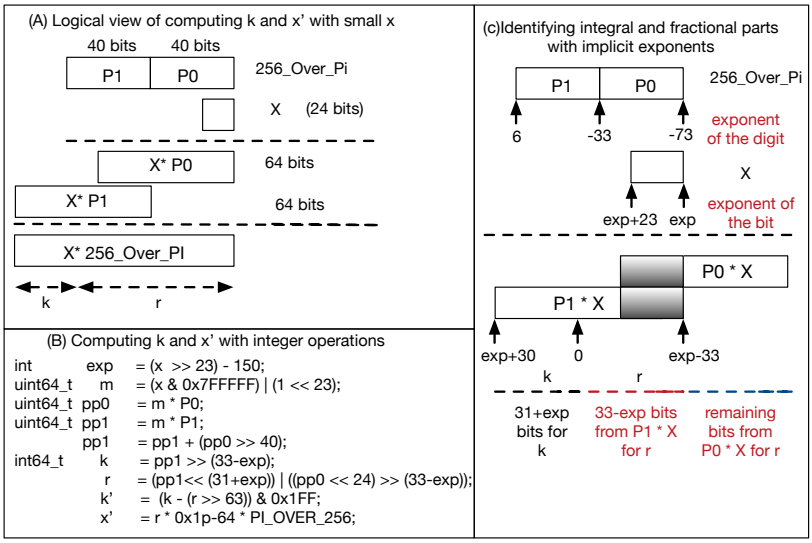

Figure 6: Range reduction for small inputs when $\frac{\pi}{128} \leq |x| < 2^{30}$ and $t = 8$. Here, we are creating a reduced input $x' \in [-\frac{\pi}{512}, \frac{\pi}{512})$. (A) Computation with 80 bits of x * 256_Over_Pi with 40 bits each in two 64-bit integers P1 and P0. (B) Computing k and r with integer and bitwise operations. (c) Intuition for finding the bit whose exponent is 0 when we perform computation with integers and do not explicitly maintain exponents.

The final task is to identify the position of the binary point in the resultant product, which will help us identify k and the bits belonging to the reduced input x'. Figure 6(c) pictorially provides the intuition

for finding the position of the binary point. The exponent of the most significant bit of $\frac{256}{\pi}$ in $P1$ is 6. The exponent of the least significant bit in $P1$ is -33. When we compute $P1 * x$, the exponent of the least significant bit of this partial product is $exp - 33$, where exp represents the exponent of the least significant bit of the original input. Since the partial product can have up to 64 precision bits, the exponent of the most significant bit would be at most $exp - 33 + 63 = exp + 30$. The binary point (*i.e.*, exponent 0) lies within this partial product (*i.e.*, between the exponents $exp + 30$ and $exp - 33$). We can compute the integral part of the product (*i.e.*, k) with a right shift operation by $33 - exp$ because there are $0 - (exp - 33) = 33 - exp$ bits after exponent 0. Next, we accumulate all bits past the binary point in the products $P1 * x$ and $P0 * x$ into a 64-bit integer r. We first need to place the $33 - exp$ fractional bits in the partial product $P1 * x$ as the most significant bits of r. We accomplish this with a left shift by $31 + exp$ because there are $64 - (33 - exp)$ non-fractional bits in $P1 * x$ pertaining to the integer portion k. Then, we need to collect the remaining bits for the reduced input from $P0 * x$. The leading 24-bits of $P0 * x$, which are added to the partial product $P1 * x$, can be removed with a left shift by 24. We subsequently place the most significant portions of the remaining bits of $P0 * x$ (*i.e.*, $(P0 * x) << 24$) after the $33 - exp$ initial bits in r obtained from $P1 * x$. This can be accomplished by performing a right shift on $(P0 * x) << 24$ with a shift amount of $33 - exp$. Finally, we compute the reduced input by multiplying r with the floating point value 2^{-64} and $\frac{\pi}{256}$ as shown in Figure 6(B).

Range reduction with large inputs. When the input $|x| > 2^{30}$, we use 192 bits of $\frac{256}{\pi}$ maintained as three 64-bit integers. We also maintain the 24 bits in the significand of input x as a 64-bit integer. We compute the partial products between x and 64-bit pieces of $\frac{256}{\pi}$ and represent them as 128-bit integers as shown in Figure 7(A). After computing the partial products, we identify the position of the binary point based on the exponent of x and the exponents of each piece of $\frac{256}{\pi}$. Figure 7(B) provides the bitwise operations that identify k and fractional bits required for the reduced input. The exponent of the most significant bit of $\frac{256}{\pi}$ is 6 and the the exponent of the least significant bit of P2, the first piece of $\frac{256}{\pi}$, is -57. When the exponent of the least significant bit of x, represented by exp, is less than 57, the integer bits pertaining to k will be completely contained in the partial product `pp2`. The remaining step involves extracting 64 fractional bits for the reduced input through bitwise shifts akin to those performed on the smaller inputs in Figure 6(B).

When exp is equal to 57, there is no need to perform any shift operations. Here, the relevant bits for k are the lower 64-bits of the partial product `pp2` and the fraction bits for the reduced input are the lower 64-bits of partial product `pp1`. When exp is greater than 57, k will be computed using the lower 64-bits of the partial product `pp2` and portions of `pp1`. The reduced input will be computed using the lower 64-bits of the partial product `pp1` and portions of `pp0`. Everything is computed with integers prior to multiplying r with 2^{-64} and $\frac{\pi}{256}$ to generate the reduced input as shown in Figure 7(B). All shift amounts in Figure 7(B) are positive and less than 64, which avoids undefined behavior. The largest value of exp in Figure 7(B), which represents the exponent of the least significant bit of the input, is 104. This is because the largest exponent of any 32-bit FP number is 127.

4 Experimental Evaluation

We have developed various implementations of trigonometric functions that produce correctly rounded results for all inputs for all FP representations up to 32-bits. Our prototype uses the MPFR library [12] as the oracle to generate the library. We developed new range reduction techniques, inference techniques for reduced intervals, and new polynomials for the trigonometric functions with the RLIBM approach. To evaluate our functions for correctness and performance, we compare it against mainstream libraries

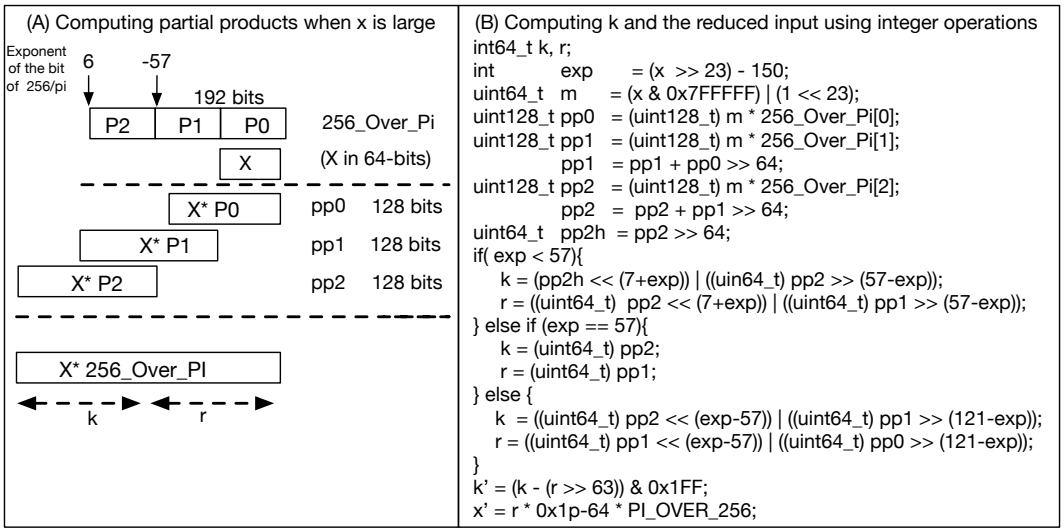

Figure 7: Range reduction for large inputs when $2^{30} \leq |x|$ when $t = 8$ to produce the reduced input $x' \in [-\frac{\pi}{512}, \frac{\pi}{512})$. (a) We show the logical computation of $\frac{256x}{\pi}$. (b) The computation with integer and bitwise operations. Each piece of 256_Over_Pi (*i.e.*, P2, P1,and P0) is 64-bits.

(*e.g.*, GLIBC's float and double libm) and the correctly rounded Core-Math library [25]. We conducted our experiments on a 2.10GHz Intel Xeon(R) Silver 4310 server with 256GB of RAM running Ubuntu 24.04.1 LTS and used performance counters to measure the time taken.

Ability to produce correct results. Our functions produce correctly rounded results for all representations from 10-bits to 32-bits for all inputs, which we tested with complete enumeration similar to bounded model checking. In contrast, GLIBC's float libm does not produce correctly rounded results for 32-bit float inputs for *sin*, *cos*, and *tan* even for one representation and has several thousand incorrect results. GLIBC's double libm is more accurate than GLIBC's float libm but still produces wrong results for some inputs. For the 32-bit float representation, both our functions and Core-Math produce correctly rounded results for all inputs. However, Core-Math does not produce correctly rounded results for all inputs from 10-bits to 32-bits, which is due to double-rounding errors.

Performance due to our range reduction optimizations. For the FP-based approach, we developed two implementations for *sin*, *cos*, and *tan* that apply the range reduction algorithms shown in Figure 5(a) and Figure 5(b). We have also developed implementations for these functions that use the integer-based range reduction depicted in Figures 6 and 7. Lastly, we experimented with applying different approaches for different sub-domains. Specifically, we implemented versions of *sin*, *cos*, and *tan* that use the FP-based approach for smaller inputs (see Algorithm 1b) and an integer-based approach for larger inputs (see Figure 7).

Among these strategies, applying a FP-based approach for smaller inputs and an integer-based approach for larger inputs leads to the best performance for all three functions. Figure 8a reports the speedup of this approach over the FP-based approach detailed in Figure 5(a) (**FP V1**), the alternative FP-based approach in Figure 5(b)(**FP V2**), and the integer-based approach (**Int**). On average, our most efficient, hybrid range reduction achieves a 19% speedup against the initial FP-based approach. This result highlights the advantage of employing an integer-based range reduction for large inputs. We attribute this speedup to the larger amount of precision relative to 64-bit doubles (*i.e.* 53-bits of precision)

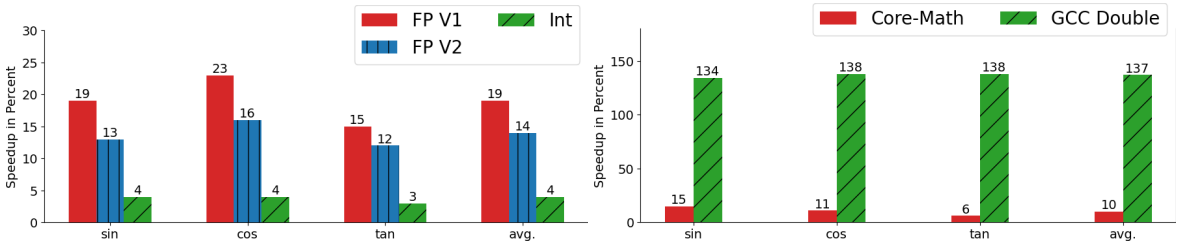

Figure 8: (a) Performance improvement using our approach that uses FP-based range reduction for small inputs and integer-based range reduction for larger inputs over other approaches. (b) Performance improvement of the fastest RLIBM implementation over other libraries.

available with integer representations (*i.e.*, `uint64_t` and `uint128_t`). The larger amount of precision available for each piece of $\frac{256}{\pi}$ and intermediate outputs in the integer-based approach reduces the number of partial products required to obtain a reduced input with sufficient accuracy. Moreover, the bit-wise operations employed for the integer-based approach greatly simplifies identifying the portions of $\frac{256x}{\pi}$ relevant to the reduced input and the lower order bits of k for any given input. Alternatively, the speedup of the hybrid approach over the integer-based approach (4% on average) indicates that FP operations provide a more ideal solution when the inputs have low magnitudes and the relevant portions of $\frac{256}{\pi}$ are confined to the initial set of bits. We attribute this result to the observation that the integer portion of $\frac{256x}{\pi}$ for small inputs (*i.e.*, $|x| < 2^{30}$) are small enough to be exactly representable using a single double-precision value. For such cases, the results indicate that computing the first few partial products of $\frac{256x}{\pi}$ and subtracting away integer bits identified through FP rounding operations are sufficient for obtaining reduced inputs with the desirable level of accuracy with low overhead.

Figure 8b reports the performance speedup of our fastest implementations when compared to other libraries. On average, our functions are 10% and 137% faster than Core-Math's functions and GLIBC's double functions respectively. The performance speedups over Core-Math are smaller because Core-Math's implementations are also well optimized with a range reduction strategy appropriately mixing both FP and integer-based approaches. Unlike Core-Math's functions, which produce correct results only for 32-bits, our functions produce correct results for multiple representations from 10-bits to 32-bits and all five rounding modes in the IEEE standard.

5 Conclusion and Future Work

We extend the RLIBM approach to trigonometric functions by paying careful attention to the amount of precision required in handling $\frac{256}{\pi}$, while obtaining performance using integer and bitwise operations. We have been collaborating with the developers of mainstream math libraries to incorporate these methods, which will enable push-button usage. We are also participating in the standards committees to mandate correct rounding in the next version of the IEEE-754 standard. Finally, we want to explore correctly rounded libraries for GPU platforms in the future.

6 Acknowledgments

We thank the VSS reviewers and Bill Zorn for their feedback. This material is based upon work supported in part by the research gifts from the Intel Corporation and by the National Science Foundation with

grants: 2110861, 2312220, and 1908798.

References

[1] Mridul Aanjaneya, Jay P. Lim & Santosh Nagarakatte (2022): *Progressive Polynomial Approximations for Fast Correctly Rounded Math Libraries*. In: *43rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'22, doi:10.1145/3519939.3523447.

[2] Mridul Aanjaneya & Santosh Nagarakatte (2024): *Maximum Consensus Floating Point Solutions for Infeasible Low-Dimensional Linear Programs with Convex Hull as the Intermediate Representation*. *Proc. ACM Program. Lang.* 8(PLDI), doi:10.1145/3656427. Available at `https://doi.org/10.1145/3656427`.

[3] Sylvie Boldo, Marc Daumas & Ren-Cang Li (2009): *Formally Verified Argument Reduction with a Fused Multiply-Add*. In: *IEEE Transactions on Computers*, 58, pp. 1139–1145, doi:10.1109/TC.2008.216.

[4] Sylvie Boldo & Guillaume Melquiond (2005): *When double rounding is odd*. In: *17th IMACS World Congress*, Paris, France, p. 11.

[5] Nicolas Brisebarre & Sylvvain Chevillard (2007): *Efficient polynomial L^∞-approximations*. In: *18th IEEE Symposium on Computer Arithmetic (ARITH '07)*, doi:10.1109/ARITH.2007.17.

[6] Nicolas Brisebarre, Jean-Michel Muller & Arnaud Tisserand (2006): *Computing Machine-Efficient Polynomial Approximations*. In: *ACM ACM Transactions on Mathematical Software*, 32, Association for Computing Machinery, New York, NY, USA, p. 236–256, doi:10.1145/1141885.1141890.

[7] Sylvain Chevillard, Mioara Joldes & Christoph Lauter (2010): *Sollya: An Environment for the Development of Numerical Codes*. In: *Mathematical Software - ICMS 2010, Lecture Notes in Computer Science* 6327, Springer, Heidelberg, Germany, pp. 28–31, doi:10.1007/978-3-642-15582-6_5.

[8] William J Cody & William M Waite (1980): *Software manual for the elementary functions*. Prentice-Hall series in computational mathematics, Prentice-Hall, Englewood Cliffs, NJ, doi:https://doi.org/10.1137/1024023.

[9] Catherine Daramy, David Defour, Florent Dinechin & Jean-Michel Muller (2003): *CR-LIBM: A correctly rounded elementary function library*. In: *Proceedings of SPIE Vol. 5205: Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, 5205, doi:10.1117/12.505591.

[10] Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Lauter & Jean-Michel Muller (2006): *CR-LIBM A library of correctly rounded elementary functions in double-precision*. Research Report, Laboratoire de l'Informatique du Parallélisme. Available at `https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804`.

[11] Marc Daumas, Christophe Mazenc, Xavier Merrheim & Jean-Michel Muller (1996): *Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions*. In: *J. UCS The Journal of Universal Computer Science*, Springer, pp. 162–175, doi:10.1007/978-3-642-80350-5_15.

[12] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier & Paul Zimmermann (2007): *MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding*. *ACM Transactions on Mathematical Software* 33(2), doi:10.1145/1236463.1236468.

[13] David Goldberg (1991): *What Every Computer Scientist Should Know About Floating-point Arithmetic*. In: *ACM Computing Surveys*, 23, ACM, New York, NY, USA, pp. 5–48, doi:10.1145/103162.103163.

[14] William Kahan (2004): *A Logarithm Too Clever by Half*. Available at `https://people.eecs.berkeley.edu/~wkahan/LOG10HAF.TXT`.

[15] Vincent Lefèvre & Jean-Michel Muller (2001): *Worst Cases for Correct Rounding of the Elementary Functions in Double Precision*. In: *15th IEEE Symposium on Computer Arithmetic*, Arith '01, pp. 111–118, doi:10.1109/ARITH.2001.930110.

[16] Vincent Lefèvre, Jean-Michel Muller & Arnaud Tisserand (1998): *Toward correctly rounded transcendentals*. *IEEE Transactions on Computers* 47(11), pp. 1235–1243, doi:10.1109/12.736435.

[17] Jay P. Lim, Mridul Aanjaneya, John Gustafson & Santosh Nagarakatte (2021): *An Approach to Generate Correctly Rounded Math Libraries for New Floating Point Variants*. Proceedings of the ACM on Programming Languages 6(POPL), doi:10.1145/3434310.

[18] Jay P. Lim & Santosh Nagarakatte (2021): *High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations*. In: 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'21, doi:10.1145/3453483.3454049.

[19] Jay P. Lim & Santosh Nagarakatte (2022): *One Polynomial Approximation to Produce Correctly Rounded Results of an Elementary Function for Multiple Representations and Rounding Modes*. Proceedings of the ACM on Programming Languages 6(POPL), doi:10.1145/3498664.

[20] Peter Markstein (2000): *IA-64 and Elementary Functions: Speed and Precision*. Prentice Hall.

[21] Jean-Michel Muller (2016): *Elementary Functions: Algorithms and Implementation*. Sprinder, 3rd edition, doi:10.1007/978-1-4899-7983-4.

[22] K. C. Ng (1992): *Argument reduction for huge arguments: Good to the last bit*.

[23] NVIDIA (2020): *TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x*. Available at `https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/`.

[24] Mary H. Payne & Robert N. Hanek (1983): *Radian Reduction for Trigonometric Functions*. SIGNUM Newsl. 18(1), p. 19–24, doi:10.1145/1057600.1057602. Available at `https://doi.org/10.1145/1057600.1057602`.

[25] Alexei Sibidanov, Paul Zimmermann & Stéphane Glondu (2022): *The CORE-MATH Project*. In: *ARITH 2022 - 29th IEEE Symposium on Computer Arithmetic*, virtual, France, doi:10.1109/ARITH54963.2022.00014. Available at `https://hal.inria.fr/hal-03721525`.

[26] Shane Story & Ping Tak Peter Tang (1999): *New algorithms for improved transcendental functions on IA-64*. In: Proceedings 14th IEEE Symposium on Computer Arithmetic, pp. 4–11, doi:10.1109/ARITH.1999.762822.

[27] The LLVM libc team (2022): *LLVM's Range Reduction for Trigonometric Functions*. `https://github.com/llvm/llvm-project/blob/main/libc/src/__support/math/range_reduction.h`.

[28] Julio Villalba, Tomas Lang & Mario A Gonzalez (2006): *Double-residue modular range reduction for floating-point hardware implementations*. IEEE Transactions on Computers 55(3), pp. 254–267, doi:10.1109/TC.2006.38.

[29] Shibo Wang & Pankaj Kanwar (2019): *BFloat16: The secret to high performance on Cloud TPUs*. Available at `https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus`.

[30] Abraham Ziv (1991): *Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit*. ACM Transactions on Mathematical Software 17(3), p. 410–423, doi:10.1145/114697.116813. Available at `https://doi.org/10.1145/114697.116813`.