Simultaneous floating-point sine and cosine for VLIW integer processors

Claude-Pierre Jeannerod, Jingyan Jourdan-Lu

Simultaneous floating-point sine and cosine for VLIW integer processors

Claude-Pierre Jeannerod
INRIA
Laboratoire LIP (CNRS, ENSL, INRIA, UCBL),
Université de Lyon, France
claude-pierre.jeannerod@ens-lyon.fr

Jingyan Jourdan-Lu*
STMicroelectronics
Compilation Expertise Center, Grenoble, France
jingyan.jourdan-lu@st.com
* *also a member of Laboratoire LIP*

***Abstract*—Graphics and signal processing applications often require that sines and cosines be evaluated at a same floating-point argument, and in such cases a very fast computation of the pair of values is desirable. This paper studies how 32-bit VLIW integer architectures can be exploited in order to perform this task accurately for IEEE single precision. We describe software implementations for `sinf`, `cosf`, and `sincosf` over $[-\pi/4, \pi/4]$ that have a proven 1-ulp accuracy and whose latency on STMicroelectronics' ST231 VLIW integer processor is 19, 18, and 19 cycles, respectively. Such performances are obtained by introducing a novel algorithm for simultaneous sine and cosine that combines univariate and bivariate polynomial evaluation schemes.**

***Keywords*-VLIW integer processor; instruction level parallelism; C software implementation; floating-point arithmetic; IEEE 754; unit in the last place; trigonometric function;**

I. Introduction

The ST231 is a 4-way integer VLIW core from STMicroelectronics' ST200 family derived from the Lx technology platform [1], which is designed to implement advanced audio and video codecs in consumer devices such as set-top boxes for HD-IPTV (High Definition Internet Protocol Television), smart phones, and wireless terminals.

As this processor has integer-only register file and ALUs, all the floating-point support is implemented by software emulation, such as the highly optimized FLIP 1.0 library [2] for IEEE single precision. This brings basic arithmetic at low cost, the latencies of $\pm$, $\times$, $/$, $\sqrt{}$ being respectively of 26, 21, 34, 23 cycles for rounding 'to nearest even' and with subnormal support.

In this paper we consider the sine and cosine functions, with special emphasis on their *simultaneous* computation, as in [3]. Typically, the evaluation of any of these functions is decomposed into three steps [4]: range reduction, which computes $x^* \in [-\pi/4, \pi/4]$ and $k \in \mathbb{Z}$ such that $x^* = x - k\pi/2$; evaluation of $\sin x^*$ or $\cos x^*$ depending on the value of $k \bmod 4$; sign adaptation to reconstruct the result. Clearly, range reduction can be shared by both functions, so that we focus here on the reduced range $[-\pi/4, \pi/4]$.

When multipliers are available on the architecture, CORDIC-like shift-and-add methods need not be used and the computation of sine and/or cosine eventually often relies on the evaluation of one or several univariate polynomial approximations, be it in hardware [5], [6], [7] or in software [8], [9], [10], [11], [12], [13], [14]. In particular, fixed-point univariate polynomials have been already employed for sine and cosine on ST231 [11, §14], yielding respective latencies of 29 and 36 cycles, without accuracy guarantee.

In this paper we propose a new approach based on the combination of fixed-point univariate and bivariate polynomials for more ILP exposure, along with rigorous accuracy analyzes. More precisely, our contributions are as follows:

First, we introduce an algorithm for sine (resp. cosine) that uses a single bivariate (resp. univariate) polynomial approximation evaluated in a highly parallel fashion, and whose accuracy is shown to be $\leqslant 1$ ulp of the exact result.

Second, we deduce from these two algorithms a third one for simultaneous sine and cosine, whose hybrid univariate/bivariate nature brings high ILP exposure.

Third, we detail the corresponding C implementations, showing optimal schedules and very low latencies on a VLIW integer processor like the ST231: 19 cycles for `sinf`, 18 for `cosf`, and 19 for `sincosf`.

Thus, compared with [11] speedups of $> 1.5\times$ for `sinf` and $2\times$ for `cosf` are obtained, together with proven 1-ulp accuracy. Also, `sincosf` yields both 1-ulp accurate sine and cosine in the same latency as it takes to compute sine alone.

Although our codes have been optimized with the ST231 architecture in mind, they are written in standard C and are thus portable.[1] In addition, the design has been assisted by a software toolchain consisting of Sollya [15] for certified polynomial approximants and of Gappa [16] and CGPE [17] to guarantee the numerical and cost features of polynomial evaluations. Finally, our approach supports subnormals for free and can easily be adapted to other floating-point formats like double precision on 64-bit VLIW integer processors.

Outline. The paper is organized as follows. §II gives an overview of the ST231 processor and compiler and, on the other hand, some reminders about IEEE single precision (also called binary32) and units in the last place (ulps). Then §§III and IV present our algorithms and implementations for cosine and sine, respectively; in each case we provide

[1] An archive containing C code for the functions `sinf` , `cosf`, `sincosf` and the assembly code produced for ST231 is available upon request.

a theoretical analysis yielding a proven 1-ulp error bound, as well as implementation details and C codes.[2] In §V we describe our simultaneous computation of sine and cosine and analyze its performances on ST231. We conclude in §VI.

II. Background and notation

A. Overview of the ST231 processor and its compiler

The ST231 is a 32-bit four-way integer VLIW core. Up to four independent instructions can be bundled into a *syllable*, achieving the maximum throughput of four instructions per cycle.

The architecture of the ST231 (depicted in Figure 1) includes the following features:

- Parallel execution units, including multiple integer ALUs and two 32×32-bit multipliers;
- A large register file of 64 32-bit general purpose registers and 8 1-bit condition registers;
- Predicated execution through 'select' operations;
- Efficient branch architecture with multiple condition registers;
- Encoding of immediate operands up to 32 bits.

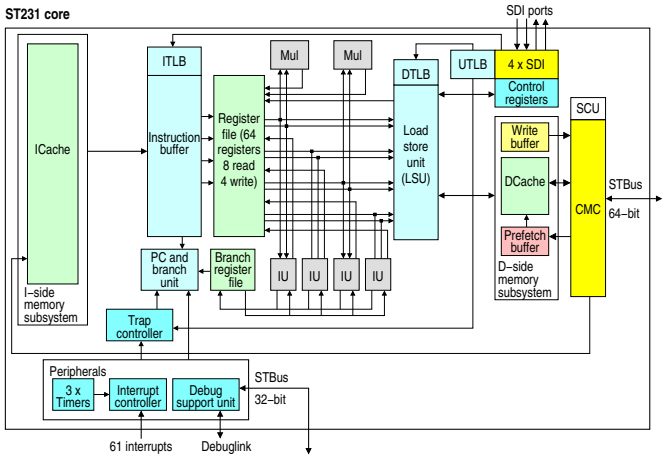

Figure 1. Architecture of the ST231 processor core.

All these features are key to our sine and cosine implementations, that use efficiently all the resources exposed by the machine. Particularly, for fast fixed-point polynomial evaluation, we must encode the 32-bit coefficients inside the instruction words to avoid costly external memory access and achieve efficient multiplication and addition schemes. The two 32×32-bit multipliers are fully pipelined and can be used simultaneously. The latency of the multipliers is 3 cycles while for all the other integer arithmetic instructions, it is only 1 cycle.

[2]The symmetry properties $\cos(-x) = \cos x$ and $\sin(-x) = -\sin x$ allow us to restrict our accuracy analyzes to nonnegative inputs, but the codes do handle the range $[-\frac{\pi}{4}, \frac{\pi}{4}]$.

The ST231 VLIW compiler, st200cc, is based on the Open64 technology retargeted for the ST200 processor family by STMicroelectronics since 2001. The compiler has been improved to support development for high performance embedded targets.

As opposed to out-of-order super-scalar processors designs that benefit from dynamic instruction dispatch on multiple execution units, VLIW processors are very dependent on compilers to statically extract ILP from the source code. This is achieved by high level transformations on the language-independent intermediate representation and transformations at the code generation level such as if-conversion [18] that converts control-flow into predicated instructions and global code motion. Then, a precise model of the machine resources exploited by several passes of scheduling, before and after register allocation, enables an efficient bundling, finally achieving a high ILP, even in non-cyclic contexts [19].

The observation of the fact that sine and cosine are very frequently called together with the same argument is not new, and indeed these patterns of code are frequent in graphics and signal processing software.

Many compilers implement the recognition of this idiom and transform it into a specific code sequence that computes a pair of results and select the appropriate value.

In our case this transformation can be done regardless of cost considerations since we can get one computation for free. The compiler replaces all the sine and cosine by a call to a specific function returning a pair of floats. As sine and cosine are 'pure' functions known by the compiler not to have any side-effect, when both results are required for the same input, the specific function will be called only once after redundancy elimination.

B. Binary32 floating-point numbers

Here and hereafter $\mathbb{F}$ denotes the set of binary32 finite floating-point numbers defined by the IEEE 754-2008 standard [20]: for

$$p = 24 \quad \text{and} \quad e_{\max} = 1 - e_{\min} = 127,$$

this set $\mathbb{F}$ is the subset of $\mathbb{R}$ consisting of ± 0, of *subnormal* numbers $\pm m \cdot 2^{e_{\min}}$ with $m = (0.m_1 \dots m_{p-1})_2$ nonzero, and of *normal* numbers $\pm m \cdot 2^e$ with $m = (1.m_1 \dots m_{p-1})_2$ and $e \in \{e_{\min}, \dots, e_{\max}\}$.

Useful values associated to $\mathbb{F}$ are the *unit roundoff*

$$u = 2^{-p}$$

(see for example [21, p. 38]), as well as the smallest positive subnormal $\alpha = 2u \cdot 2^{e_{\min}}$ and the largest finite number $\Omega = (2-2u) \cdot 2^{e_{\max}}$. We say that a real x belongs to the subnormal range of $\mathbb{F}$ when $0 < |x| < 2^{e_{\min}}$, and to the normal range of $\mathbb{F}$ when $2^{e_{\min}} \leqslant |x| \leqslant \Omega$. Note also that the significand m of any normal number satisfies $1 \leqslant m \leqslant 2 - 2u$.

Furthermore, the standard [20] associates to each number $x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}$ in $\mathbb{F}$ a unique *encoding* into a 32-bit unsigned integer X: the bit string of X is

$$[s_x | E_{x,7} \cdots E_{x,0} | m_{x,1} \cdots m_{x,23}], \tag{1}$$

where the bits $E_{x,i}$ define the so-called *biased exponent* $E_x = \sum_{i=0}^{7} E_{x,i} 2^i = e_x + m_{x,0} - e_{\min}$; the respective encodings of $+0$ and -0 are 0 and 2^{31}.

C. Unit in the last place (ulp): definition and properties

Units in the last place (ulps) are often used to indicate the accuracy of floating-point results. Given a floating-point system of precision p and minimal exponent $e_{\min}$, the ulp of any real number x can be defined as follows [22], [23]:

$$\mathrm{ulp}(x) = \begin{cases} 0 & \text{if } x = 0, \\ 2^{\max\{e_{\min}, e\} - p + 1} & \text{otherwise,} \end{cases}$$

with e the integer power of two such that $2^e \leqslant |x| < 2^{e+1}$. Let us also recall three basic properties of the ulp function, that are easy to check and will be useful in the sequel:

- For $x, y \in \mathbb{R}$,

$$|x| \leqslant |y| \quad \Rightarrow \quad \mathrm{ulp}(x) \leqslant \mathrm{ulp}(y). \tag{2a}$$

- If $j \in \mathbb{Z}$ and $x \in \mathbb{R}$ are such that both x and $2^j x$ are in the normal range of $\mathbb{F}$ then

$$\mathrm{ulp}(2^j x) = 2^j \mathrm{ulp}(x). \tag{2b}$$

- For $x \in \mathbb{R}$,

$$0 < |x| < 2^{e_{\min}+1} \quad \Rightarrow \quad \mathrm{ulp}(x) = \alpha. \tag{2c}$$

Finally, thanks to the above definition, "1-ulp accuracy" means that our implementations conform to the following precise specification: assuming an input x in

$$I = \mathbb{F} \cap [0, \pi/4],$$

for cosine we want an r in $\mathbb{F}$ such that

$$|r - \cos x| \leqslant \mathrm{ulp}(\cos x). \tag{3}$$

Similarly, for sine this means an r in $\mathbb{F}$ such that

$$|r - \sin x| \leqslant \mathrm{ulp}(\sin x). \tag{4}$$

III. Computing cosine

Over $[0, \pi/4]$ the cosine function is strictly decreasing and takes values between 1 and $1/\sqrt{2} \approx 0.707$. Thus, for $x \in I$,

$$\mathrm{ulp}(\cos x) = \begin{cases} 2u & \text{if } x = 0, \\ u & \text{otherwise,} \end{cases}$$

and to satisfy our accuracy requirement in (3) it suffices to ensure

$$|r - \cos x| \leqslant u. \tag{5}$$

A. Constant approximation when $x < 2^{-11}$

Of course, when x is "close enough" to zero then $\cos x$ will be "close enough" to one and can be approximated by a constant in $\mathbb{F}$. More precisely, by Taylor's theorem, for $x > 0$ there exists a real x_0 in $(0, x)$ such that

$$\cos x = 1 - \tfrac{\cos x_0}{2} x^2. \tag{6}$$

Hence $0 \leqslant 1 - \cos x \leqslant \frac{1}{2} x^2$ and the accuracy condition in (5) holds with $r = 1 - u$ as long as $x \leqslant 2\sqrt{u}$. For binary32 this threshold is equal to 2^{-11}.

Thus, in principle we would start by detecting whether x satisfies $x \leqslant 2^{-11}$ or not. However, we prefer to replace that condition by the *strict* inequality $x < 2^{-11}$, which is just slightly stronger but has the advantage of being equivalent to deciding whether the biased exponent of x satisfies $E_x < e_{\max} - 11 = 116$. Since E_x will be needed anyway when handling inputs larger than 2^{-11} (as shall be seen in Section III-B), this is a way of reusing it to filter out small inputs.

To sum up, when x is in $I_{<2^{-11}}$, that is, when $E_x < 116$ then we simply return the encoding of $1 - u$, namely

$$127 \cdot 2^{23} - 1 = (3f7fffff)_{16}.$$

This corresponds to the first and last lines of Listing 1.

Note that since $e_{\min} = -126$, this first subinterval $I_{<2^{-11}}$ contains zero as well as *all* positive subnormal numbers.

B. Polynomial approximation when $x \geqslant 2^{-11}$

Let us now see how to evaluate $\cos x$ when x is in $I_{\geqslant 2^{-11}}$. We start by proving that in this case any floating-point number r satisfying (5) has exponent equal to -1.

Lemma 1: Let $x \in I_{\geqslant 2^{-11}}$ and $r \in \mathbb{F}$ be as in (5). Then

$$\tfrac{1}{2} \leqslant r < 1.$$

Proof: By assumption, $\cos x - u \leqslant r \leqslant \cos x + u$. Since $0 \leqslant x \leqslant \pi/4$, we have $\cos x - u \geqslant \frac{1}{\sqrt{2}} - 2^{-p} \geqslant \frac{1}{2}$ as soon as $p \geqslant 3$, which is of course the case for binary32. Let us now check that $\cos x + u < 1$. Recalling that we have (6) with $x_0 < x$ and that the cosine function is decreasing on $[0, \pi/4]$, we obtain $\cos x_0 > \cos x$ and thus, for any x in I,

$$\cos x \leqslant \left(1 + \tfrac{x^2}{2}\right)^{-1}.$$

Now, this upper bound is less than $1 - u$ as soon as $x > \sqrt{2u/(1-u)}$, which is true since $x \geqslant 2^{-11} = 2\sqrt{u}$. ∎

Lemma 1 implies that our floating-point unknown $r = m_r \cdot 2^{e_r}$ is in fact a fixed-point number of the form

$$r = (0.1 m_{r,1} \ldots m_{r,23})_2. \tag{7}$$

To obtain r as in (5) and (7) we extend to cosine the approach introduced in [24] for square root and consisting in the truncation of a Q0.32 number v that approximates "accurately enough" from above the exact result. A rigorous

sufficient condition on the accuracy of v is provided by the theorem below.

Theorem 1: Let $x \in I_{\geqslant 2^{-11}}$, let $v = (0.v_1v_2 \dots v_{32})_2$ be such that

$$|v - g(x)| \leqslant \tfrac{u}{2} \quad \text{with} \quad g(x) = \tfrac{u}{2} + \cos x,$$

and let r be the truncated value of v after 24 fraction bits. Then r is a binary32 number that satisfies (5).

Proof: We have $r = (0.v_1v_2 \dots v_{24})_2$, which implies that r is a binary32 number and, since $u = 2^{-24}$, that

$$-u < r - v \leqslant 0.$$

On the other hand, v satisfies

$$0 \leqslant v - \cos x \leqslant u.$$

Hence $-u < r - \cos x \leqslant u$, from which (5) follows. ∎

To compute from x a value v as in Theorem 1 we essentially rely on polynomial approximation and evaluation. The process consists of three steps:

- Precompute a polynomial a that approximates g over $I_{\geqslant 2^{-11}}$ and whose fixed-point coefficients have at most 32 fraction bits. This approximation generates the error $\epsilon_0 = \max_{x \in I_{\geqslant 2^{-11}}} |a(x) - g(x)|$.
- Compute a 32-bit fixed-point approximation t to the floating-point input x. This approximation yields the error $\epsilon_1 = |a(t) - a(x)|$.
- Given t and the coefficients of a, compute an approximation v to $a(t)$ in 32-bit fixed-point arithmetic by choosing a suitable polynomial evaluation scheme. The associated evaluation error is $\epsilon_2 = |v - a(t)|$.

Thus, by the triangle inequality, the accuracy constraint $|v - g(x)| \leqslant \frac{u}{2}$ in Theorem 1 is satisfied as soon as we have

$$\epsilon_0 + \epsilon_1 + \epsilon_2 \leqslant \tfrac{u}{2}. \tag{8}$$

The next three paragraphs give some implementation details showing how to achieve (8) very efficiently on the ST231 architecture. A fourth paragraph details how to reconstruct the encoding of the result r directly from the encoding of v.

1. Precomputing the polynomial approximant a. A polynomial that satisfies the necessary condition $\epsilon_0 \leqslant \frac{u}{2}$ must have degree at least 6; this can be seen using the software tool Sollya [15], which further gives us:

- coefficients of the form $a_0 = 1 + A_0 \cdot 2^{-32}$, $a_4 = A_4 \cdot 2^{-32}$ and $a_i = -A_i \cdot 2^{-32}$ for $i \in \{1, 2, 3, 5, 6\}$ and where the seven A_i are 32-bit unsigned integers;
- the rigorous bound $\epsilon_0 < 2^{-28.86}$.

2. Approximating x by t. For x in $I_{\geqslant 2^{-11}}$ we have

$$x = (\underbrace{0.00\dots00}_{|e_x| \text{ zeros}} 1 m_{x,1} \dots m_{x,23})_2, \quad -11 \leqslant e_x \leqslant -1,$$

showing that the fraction of x can be up to 34-bit long. An approximation of x that will be enough for our purpose is its truncation after 32 fraction bits, that is,

$$t = \lfloor x \cdot 2^{32} \rfloor / 2^{32} = (0.t_1t_2 \dots t_{32})_2.$$

Hence $0 \leqslant x - t < 2^{-32}$ and, using the software tool Gappa [16], we get the rigorous bound $\epsilon_1 < 2^{-32.98}$.

Furthermore, this approximation is stored into the 32-bit unsigned integer $T = t \cdot 2^{32}$, which can be deduced from X in (1) as follows: shift X left by 8 places, set the leftmost bit to 1, and then shift the result right by

$$|e_x| - 1 = 126 - E_x$$

places. Lines 2 and 3 of Listing 1 give the corresponding C code, which on ST231 takes 4 cycles.

3. Evaluating $a(t)$ fast and accurately in fixed point. High ILP exposure is obtained by parenthesizing our degree-6 polynomial $a(y) = a_0 + \dots + a_6y^6$ as

$$\big((a_0 + a_1y) + (a_2 + a_3y)z\big) + \big((a_4 + a_5y) + a_6z\big)(z^2)$$

with $z = y^2$. With unbounded parallelism and latencies of 3 and 1 for $\times$ and $+$, this scheme takes 11 cycles, which is 2.18 times less than Horner's rule [25, p. 486].

The corresponding C code using (unsigned) 32-bit integer arithmetic appears at lines 4 to 14 in Listing 1. (Here `mul` denotes the multiplier available on ST231, which returns the 32 most significant bits of the exact product of two 32-bit unsigned integers.) We see that the parallelism constraints of the ST231 still allow for an 11-cycle latency. Concerning the accuracy of this scheme, we used Gappa to check that all the computed quantities are positive Q0.32 numbers and also to get a rigorous bound on $\epsilon_2 = |v - a(t)|$ with $v = V \cdot 2^{-32}$. In our case, it turns out that $\epsilon_2 < 2^{-30.04}$, so that overall (8) is satisfied.

4. Reconstructing the result. From Theorem 1 and since $V = v \cdot 2^{32}$, we conclude that the encoding of our result r is $R = 125 \cdot 2^{23} + \lfloor V/2^8 \rfloor$. Once V is available its computation thus takes 2 cycles, as line 16 of Listing 1 shows.

Listing 1
COSINE EVALUATION IN BINARY32 OVER $[-\frac{\pi}{4}, \frac{\pi}{4}]$.

```
Ex = (X >> 23) & 0xff;                                                  0
                                                                        1
mx = (X << 8) | 0x80000000;                                             2
T = mx >> (126 - Ex);                                                   3
Z = mul(T, T);          A5T = mul(A5, T);                               4
A1T = mul(A1, T);       A3T = mul(A3, T);                               5
                                                                        6
Z2 = mul(Z, Z); A45 = A4 - A5T; A6Z = mul(A6, Z);                       7
A01 = A0 - A1T; A23 = A2 + A3T;                                         8
A23Z = mul(A23, Z);                                                     9
A46Z = A45 - A6Z;                                                       10
A46 = mul(A46Z, Z2);                                                    11
A03 = A01 - A23Z;                                                       12
                                                                        13
V = A03 + A46;          // V encodes v = (0.v1...v32)                   14
                                                                        15
R = (125 << 23) + (V >> 8);                                             16
if (Ex < 116) return 0x3f7fffff; else return R;                         17
```

Remarks:

- When compiling the code in Listing 1 with st200cc, a latency of 18 cycles is achieved, thus indicating an optimal schedule: 4 cycles for `T`, 11 cycles for `V`, 2 cycles for `R`, and a final 1-cycle 'select' to choose between `R` and the constant `0x3f7fffff`.
- Also, it is not hard to see that the C code in Listing 1 works not only for the range $[0, \frac{\pi}{4}]$ but also for $[-\frac{\pi}{4}, \frac{\pi}{4}]$. This is solely due to the first line, so that for applications where x is known to be in $[0, \frac{\pi}{4}]$, one can replace this line by `Ex = X >> 23;` and thus reduce the latency by 1 cycle.

IV. Computing sine

Over $[0, \pi/4]$ the sine function is strictly increasing and takes values between 0 and $1/\sqrt{2}$. This output range contains values that vary a lot in magnitude, making the ulp analysis more involved than for cosine. A classical workaround is to consider instead of $\sin x$ the function $\frac{\sin x}{x}$ whose range for $0 < x \leqslant \pi/4$ belongs to $[0.89, 1)$. Indeed, by Taylor's theorem there exists for $x > 0$ a real x_0 in $(0, x) \subset [0, \pi/4]$ such that

$$\sin x = x - \tfrac{\cos x_0}{6} x^3, \tag{9}$$

from which it follows that for any real x in $(0, \pi/4]$,

$$x(1 - \tfrac{x^2}{6}) < \sin x < x. \tag{10}$$

This enclosure implies that $\frac{\sin x}{x}$ ranges in $[1 - \epsilon, 1)$ with $\epsilon = \frac{\pi^2}{96} = 0.102\ldots$. It also serves as a basis for the following result, which describes the behavior of the ulp of sine and is a key ingredient for establishing the numerical accuracy of our implementations.

Lemma 2: Let x be a real in $[0, \pi/4]$. Then

$$\mathrm{ulp}(\sin x) = \begin{cases} \mathrm{ulp}(x) & \text{if } x < 2^{e_{\min}+1}, \\ \frac{1}{2}\mathrm{ulp}(x) \text{ or } \mathrm{ulp}(x) & \text{otherwise.} \end{cases}$$

Proof: If $x = 0$ then $\sin x = x$ and the result is true. If $0 < x < 2^{e_{\min}+1}$ then (10) gives $\sin x < x < 2^{e_{\min}+1}$ and by using (2c) we deduce that $\mathrm{ulp}(\sin x) = \mathrm{ulp}(x) = \alpha$. It remains to consider the case where $x \geqslant 2^{e_{\min}+1}$. In this case, we have $0 < x < 1$, so that (10) leads to

$$\tfrac{x}{2} < \sin x < x$$

and thus, using (2a), to $\mathrm{ulp}(\frac{x}{2}) \leqslant \mathrm{ulp}(\sin x) \leqslant \mathrm{ulp}(x)$. On the other hand, $x \geqslant 2^{e_{\min}+1}$ implies that both $\frac{x}{2}$ and x are in the normal range of $\mathbb{F}$, and (2b) then gives $\mathrm{ulp}(\frac{x}{2}) = \frac{1}{2}\mathrm{ulp}(x)$. Consequently, $\mathrm{ulp}(\sin x)$ ranges between $\frac{1}{2}\mathrm{ulp}(x)$ and $\mathrm{ulp}(x)$; but since ulps are integer powers of two, this means that $\mathrm{ulp}(\sin x)$ must be one of these two values. ∎

A. Approximation by x when $x < 2^{-11}$

Just like for cosine, for $x \in I$ in the neighborhood of zero we use the second-order Taylor approximation of $\sin x$ and thus return $r = x$; see (9). For binary32 it turns out that this choice guarantees the accuracy condition (4) as long as $x \leqslant 2^{-11}$. (This can be checked easily using arguments similar to those employed for the proof of Lemma 2.) Also, again like for cosine, in practice we work with the strict inequality $x < 2^{-11}$ instead. Thus, the general structure of the C code for sine, shown in Listing 2, is similar to the one of Listing 1 and when $x \in I_{<2^{-11}}$ we simply return $r = x$.

B. Bivariate polynomial approximation when $x \geqslant 2^{-11}$

Assume now that our input x is in $I_{\geqslant 2^{-11}}$. Unlike for cosine and as can be expected, the exponent of the result is not anymore a constant known in advance. However, the result below shows that any floating-point number satisfying the accuracy constraint in (4) can have only two possible exponents, namely e_x or $e_x - 1$.

Lemma 3: Let $x \in I_{\geqslant 2^{-11}}$ and $r \in \mathbb{F}$ be as in (4). Then, writing e_x for the exponent of x we have

$$2^{e_x - 1} \leqslant r < 2^{e_x + 1}.$$

Proof: Since x is a normal number, we have $\mathrm{ulp}(x) = 2u \cdot 2^{e_x}$, so that (4) implies

$$\sin x - 2u \cdot 2^{e_x} \leqslant r \leqslant \sin x + 2u \cdot 2^{e_x}.$$

Furthermore, x is in $(0, \pi/4]$ and we can apply (10): the upper bound in (10) leads to $r < x + 2u \cdot 2^{e_x} = (m_x + 2u) \cdot 2^{e_x}$ and since $m_x \leqslant 2 - 2u$ we conclude that $r < 2^{e_x+1}$; using the lower bound gives $r > \big(m_x(1 - \frac{x^2}{6}) - 2u\big) \cdot 2^{e_x}$ and since $m_x \geqslant 1$ and $0 < x < 1$ we arrive at $r > (1 - \frac{1}{6} - 2u) \cdot 2^{e_x}$. For $p \geqslant 3$—which is the case for binary32—,this implies $r \geqslant 2^{e_x - 1}$ and the conclusion follows. ∎

Lemma 3 indicates that for floating-point solutions r to (4) the binary expansion of $r/2^{e_x}$ is either $(0.1 *_1 \cdots *_{23})_2$ or $(1. *_1 \cdots *_{23})_2$. In the theorem below we show how to recover such bits from truncating a suitable approximation v to a function of our input x.

Theorem 2: Let $x \in I_{\geqslant 2^{-11}}$, let $v = (v_0.v_1 \ldots v_{31})_2$ be such that

$$|v - h(x)| \leqslant \tfrac{u}{2} \quad \text{with} \quad h(x) = \tfrac{u}{2} + \tfrac{\sin x}{2^{e_x}}$$

and with e_x the exponent of x, and let r be defined as

$$r = \begin{cases} (0.v_1 \ldots v_{23}v_{24})_2 \cdot 2^{e_x} & \text{if } v < 1, \\ (1.v_1 \ldots v_{23})_2 \cdot 2^{e_x} & \text{if } v \geqslant 1. \end{cases}$$

Then r is a binary32 number satisfying (4).

Proof: Note first that $e_x > e_{\min}$, which implies that r is a binary32 number and that $\mathrm{ulp}(x) = 2u \cdot 2^{e_x}$. Note also that the definition of h gives

$$0 \leqslant v \cdot 2^{e_x} - \sin x \leqslant u \cdot 2^{e_x}. \tag{11a}$$

If $v < 1$ then $r/2^{e_x}$ is the truncated value of v after 24 fraction bits, so that

$$-u \cdot 2^{e_x} < r - v \cdot 2^{e_x} \leqslant 0. \tag{11b}$$

Using (11a) and (11b) leads to the upper bound $|r - \sin x| \leqslant u \cdot 2^{e_x} = \frac{1}{2}\mathrm{ulp}(x)$, which by Lemma 2 is at most $\mathrm{ulp}(\sin x)$. Hence (4) is satisfied when $v < 1$.

Assume now that $v \geqslant 1$. In this case, $r/2^{e_x}$ is the truncated value of v after 23 fraction bits and thus

$$-2u \cdot 2^{e_x} < r - v \cdot 2^{e_x} \leqslant 0. \tag{11c}$$

We consider two subcases separately, depending on the value of the ulp of sine:

- If $\mathrm{ulp}(\sin x) = \mathrm{ulp}(x)$ then (4) is equivalent to $|r - \sin x| \leqslant 2u \cdot 2^{e_x}$, which holds thanks to (11a) and (11c).
- If $\mathrm{ulp}(\sin x) = \frac{1}{2}\mathrm{ulp}(x)$ then $\sin x < 2^{e_x} \leqslant x$, so that the upper bound in (11a) leads to

  $$1 \leqslant v < 1 + u = (1.\underbrace{00\ldots00}_{\text{23 zeros}}1)_2.$$

  Since by assumption v is a Q0.32 number, this implies $v_1 = \cdots = v_{23} = 0$ and then $r = 2^{e_x}$. Therefore, in this subcase, (4) is equivalent to $2^{e_x} - \sin x \leqslant u \cdot 2^{e_x}$, which holds true thanks to (11a) and since $v \geqslant 1$.

Thus (4) holds for $v \geqslant 1$ too, which concludes the proof. ■

Theorem 2 is obviously the counterpart of Theorem 1, that we have established for cosine in Section III-B. However, its efficient implementation by means of polynomial approximation and evaluation is somehow more delicate, since the function h does not only depend on $\sin x$ but also on 2^{e_x}.

An efficient solution is to generalize to our context the bivariate polynomial approach of [24]: writing m_x for the significand of x, we have $\frac{\sin x}{2^{e_x}} = m_x \cdot \frac{\sin x}{x}$ and we can view $h(x)$ as the value at (m_x, x) of the bivariate function

$$\widetilde{h}(s, y) = \tfrac{u}{2} + s \cdot \tfrac{\sin y}{y}. \tag{12}$$

We then follow the same three-step process as seen for cosine, but with a polynomial in two variables instead of just one:

- Precompute a suitable bivariate polynomial $\widetilde{b}$ that approximates $\widetilde{h}$ over $[1, 2) \times I_{\geqslant 2^{-11}}$.
- In the evaluation pair (m_x, x), only x may not fit into 32 bits, so we approximate it by t as for cosine.
- Given m_x, t, and the coefficients of $\widetilde{b}$, compute an approximate value v to $\widetilde{b}(m_x, t)$.

This process generates three errors, say $\widetilde{\epsilon}_0, \widetilde{\epsilon}_1, \widetilde{\epsilon}_2$ (defined essentially as for cosine by replacing a and g with, respectively, $\widetilde{b}$ and $\widetilde{h}$). In order to apply Theorem 2 it suffices that the following analogue of (8) be satisfied:

$$\widetilde{\epsilon}_0 + \widetilde{\epsilon}_1 + \widetilde{\epsilon}_2 \leqslant \tfrac{u}{2}. \tag{13}$$

We conclude this section by detailing our implementation choices for those stages as well as for the reconstruction of the result.

1. Precomputing the bivariate polynomial approximant $\widetilde{b}$. Due to the shape of $\widetilde{h}$ in (12) we use a special bivariate polynomial of the form

$$\widetilde{b}(s, y) = \tfrac{u}{2} + s \cdot b(y).$$

Consequently,

$$\left|\widetilde{b}(s, y) - \widetilde{h}(s, y)\right| = s \cdot \Delta \quad \text{with} \quad \Delta = \left|b(y) - \tfrac{\sin y}{y}\right|,$$

and since $1 \leqslant s < 2$, we search for a polynomial b of minimal degree such that $\Delta \leqslant \frac{1}{2} \cdot \frac{u}{2} = 2^{-26}$. Using again Sollya, an even polynomial of degree 6 is found:

$$b(y) = b_0 + b_2 y^2 + b_4 y^4 + b_6 y^6,$$

where, for $0 \leqslant i \leqslant 3$, $b_{2i} = (-1)^i \cdot B_{2i}$ and B_{2i} is a 32-bit unsigned integer. In addition, $\Delta < 2^{-27.84}$ and thus $\widetilde{\epsilon}_0 < 2^{-26.84}$.

2. Approximating x by t. Our C code for this step is the same as for cosine and appears at lines 2 and 3 of Listing 2. Again, we have $0 \leqslant x - t < 2^{-32}$ and using Gappa now leads to $|b(t) - b(x)| < 2^{-34.57}$ and thus to $\widetilde{\epsilon}_1 < 2^{-33.57}$.

3. Evaluating $\widetilde{b}(m_x, t)$ fast and accurately in fixed point. A highly parallel scheme for our bivariate polynomial is

$$\widetilde{b}(s, y) = \big((2^{-25} + sb_0) + (sb_2)z\big) + \big(sb_4 + (sb_6)z\big)(z^2),$$

where $z = y^2$. On ST231 this scheme takes 11 cycles and using the software tool CGPE [17], one can check that even with the knowledge that $s = m_x$ is available earlier than $y = t$ this latency cannot be reduced further by any other parenthesization.

A fixed-point implementation of this scheme is given in Listing 2: all the variables of the form `sB*` encode Q1.31 numbers, while as for cosine `Z` and `Z2` encode Q0.32 numbers. We have checked with Gappa the absence of overflow and that the Q1.31 number

$$v = V \cdot 2^{-31} = (v_0.v_1 \ldots v_{31})_2.$$

satisfies $\widetilde{\epsilon}_2 := |v - \widetilde{b}(m_x, t)| < 2^{-31.37}$, so that the accuracy constraint in (13) is eventually satisfied.

4. Reconstructing the result. By Lemma 3 and Theorem 2 the result is $r = (1.v_2 \ldots v_{24})_2 \cdot 2^{e_x - 1}$ if $v_0 = 0$, and $r = (1.v_1 \ldots v_{23})_2 \cdot 2^{e_x}$ if $v_0 = 1$. Therefore, its biased exponent is $E_r = E_x - 1 + v_0$ and its encoding has the form

$$R = H \cdot 2^{23} + L$$

with a high part $H = K + v_0$ such that $K = s_x \cdot 2^8 + E_x - 2$, and with a low part $L = \lfloor V/2^{7+v_0} \rfloor$.

For the cost, the bottleneck is the computation of v_0. Given V one could of course get it as `v0 = V >> 31;`. A faster way, shown at line 14 of Listing 2, consists in producing v_0 in parallel with the addition of `sB02` and `sB46` that gives V. Indeed, since `sB02` is available before `sB46`, we precompute the signed integer 2^{31} - `sB02` and compare it

with sB46 (whose first bit has been checked with Gappa to be always zero); this is equivalent to evaluating the condition $V \geqslant 2^{31}$, whose value is precisely v_0.

To summarize, this implementation brings R in 3 cycles after V. In addition, since K and H carry the sign bit of X, it works for signed inputs, that is, for any x in $\mathbb{F} \cap [-\frac{\pi}{4}, \frac{\pi}{4}]$.

Listing 2
SINE EVALUATION IN BINARY32 OVER $[-\frac{\pi}{4}, \frac{\pi}{4}]$.

```
0  Ex = (X >> 23) & 0xff;
1
2  mx = (X << 8) | 0x80000000;
3  T = mx >> (126 - Ex);      sB6 = mul(mx, B6);
4  Z = mul(T, T);             sB4 = mul(mx, B4);
5                             sB2 = mul(mx, B2);
6                             sB0 = mul(mx, B0);
7  Z2 = mul(Z, Z);            sB6Z = mul(sB6, Z);
8                             sB2Z = mul(sB2, Z);
9                             sB0_up = 0x40 + sB0;
10 sB46Z = sB4 - sB6Z;
11 sB46 = mul(sB46Z, Z2);     sB02 = sB0_up - sB2Z;
12                            J = 0x80000000 - sB02;
13                            K = (X >> 23) - 2;
14 V = sB02 + sB46;           v0 = (int32_t)sB46 >= J;
15
16 H = (K + v0) << 23;        L = (V >> 7) >> v0;
17 R = H + L;
18 if (Ex < 116) return X; else return R;
```

Remarks:

- As for cosine, the st200cc compiler is able to optimally schedule the code in Listing 2, thus leading to a latency of 4 + 11 + 3 + 1 = 19 cycles.
- One can restrict Listing 2 to the range $[0, \frac{\pi}{4}]$ by modifying only its first line (in exactly the same way as for cosine; see the remark at the end of Section III-B). This will reduce the latency from 19 to 18 cycles.

V. COMPUTING SINE AND COSINE SIMULTANEOUSLY

Given the codes for sine and cosine described and analyzed so far, an implementation of a 1-ulp accurate operator sincosf is straightforward: we essentially merge Listings 1 and 2, renaming some variables whenever necessary. The resulting C code has the form

```
unit64_t sincosf( uint32_t X ) {...}
```

and, given the encoding X of x in $\mathbb{F} \cap [-\frac{\pi}{4}, \frac{\pi}{4}]$, it returns R whose leftmost 32 bits contain the encoding of $r_1 \in \mathbb{F}$ such that $|r_1 - \sin x| \leqslant \text{ulp}(\sin x)$, and whose rightmost 32 bits encode $r_2 \in \mathbb{F}$ such that $|r_2 - \cos x| \leqslant \text{ulp}(\cos x)$.

The performances obtained by compiling the code for sincosf with the st200cc compiler (in -O3 and for the ST231 core) are summarized in Table I. The results for the restricted range $[0, \frac{\pi}{4}]$ are indicated within square brackets. (Thanks to if-conversion and the 'select' instruction, we get straight-line assembly code whose latency is independent of the value of the input.)

The main conclusion is that we get both sine and cosine in exactly the same latency as it takes to compute sine alone.

Reasons for this are the relatively low IPC of separate sine and cosine, but also the fact that several instructions

Table I
PERFORMANCES OF 1-ULP ACCURATE BINARY32 SINE, COSINE, AND SIMULTANEOUS SINE AND COSINE ON ST231.

	Latency L	Number N of instructions	IPC = N/L
sinf	19 [18]	31 [30]	1.6 [1.7]
cosf	18 [17]	25 [24]	1.4 [1.4]
sincosf	19 [18]	46 [45]	2.4 [2.5]

are common to both functions. Specifically, the number of instructions for sincosf is 10 less than the sum of those numbers for sine and cosine taken separately. On the other hand, by inspecting Listings 1 and 2 we see that sine and cosine share the computation of the biased exponent Ex and significand mx of the input, as well as the computation of T, Z, Z2 and of the predicate Ex < 116.

This said, as Table II shows, the two polynomial evaluations used within sincosf do not have much in common: they share only the computation of t^2 and t^4 (variables Z and Z2).

Table II
COMPUTATIONAL RESOURCES USED BY THE TWO POLYNOMIAL EVALUATIONS.

	sine	cosine	shared by both	total
$\times$	7	6	2	15
$+, -$	4	6	0	10
32-bit constants	4	6	0	10
9-bit constants	1	1	0	2

Figure 2 gives a precise description of the bundle occupancy when compiling sincosf with st200cc. The slots in black are those used to compute sine, those marked by $*$ are also used for cosine, and those in grey are used for cosine only. Here, every constant longer than 9 bits occupies one slot beyond the slot used by the instruction operating on it. Among the 19 bundles used, 15 are full or have 3 slots occupied. For this reason, the 4 issues of the ST231 are key to achieve a latency of 19 cycles.

VI. CONCLUDING REMARKS

We have presented here a very fast sincosf function and have also established its 1-ulp accuracy. However, in view of the intermediate error bounds used in the proofs and by replacing the truncation of polynomial values by rounding to nearest, getting closer to the 0.5-ulp ideal accuracy should be achievable with only a few extra cycles.

Conversely—and this seems more challenging, what exactly can we gain when relaxing the accuracy constraints? For example, OpenCL requires only 4 ulps of accuracy for sine and/or cosine [26], but it is not clear if a significant speed up ($> 25\%$, say) is then possible.

Finally, a third direction that we are also currently investigating deals with range reduction: so far we have assumed an input in $[-\frac{\pi}{4}, \frac{\pi}{4}]$, but for large inputs it remains to study

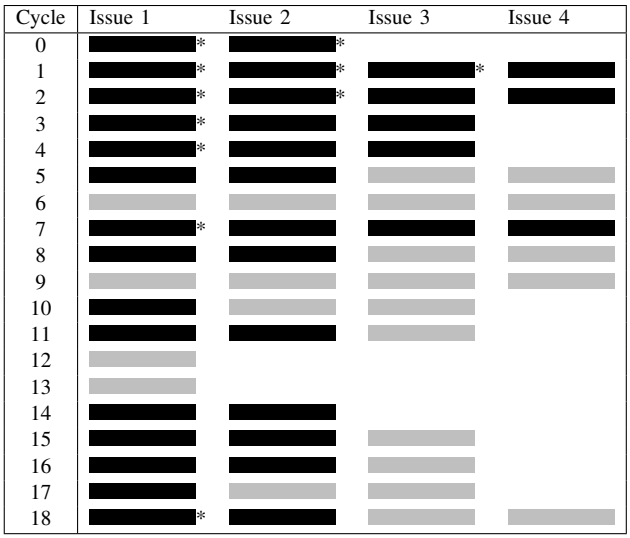

Figure 2. Bundle occupancy for the `sincosf` operator on ST231.

how reduction to that range can be performed accurately enough and very efficiently on VLIW integer architectures.

REFERENCES

[1] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: a technology platform for customizable VLIW embedded processing," in *Proc. of the 27th International Symposium on Computer Architecture (ISCA)*. ACM, 2000, pp. 203–213.

[2] C.-P. Jeannerod and G. Revy, "FLIP 1.0: a fast floating-point library for integer processors," http://flip.gforge.inria.fr/, February 2009.

[3] P. Markstein, "Accelerating sine and cosine evaluation with compiler assistance," in *Proc. IEEE Symposium on Computer Arithmetic (ARITH)*, 2003, pp. 137–140.

[4] J.-M. Muller, *Elementary Functions, Algorithms and Implementation*, 2nd ed. Birkhäuser Boston, MA, USA, 2006.

[5] V. Paliouras, K. Karagianni, and T. Stouraitis, "A floating-point processor for fast and accurate sine/cosine evaluation," *IEEE Trans. on Circuits and Systems - Part II: Analog and Digital Signal Processing*, vol. 47, no. 5, pp. 441–451, 2000.

[6] A. Tisserand, "Hardware operator for simultaneous sine and cosine evaluation," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, vol. 3, 2006, pp. 992–995.

[7] J. Detrey and F. de Dinechin, "Floating-point trigonometric functions for FPGAs," in *Field-Programmable Logic and Applications*. IEEE, 2007, pp. 29–34.

[8] S. Gal and B. Bachelis, "An accurate elementary mathematical library for the ieee floating point standard," *ACM Trans. Math. Softw.*, vol. 17, pp. 26–45, March 1991.

[9] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*, ser. Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ, 2000.

[10] M. Cornea, J. Harrison, and P. T. P. Tang, *Scientific Computing on Itanium®-based Systems*. Intel Press, 2002.

[11] S.-K. Raina, "FLIP: a Floating-point Library for Integer Processors," Ph.D. dissertation, ENS de Lyon, France, 2006.

[12] K. G. Lenzi and O. Saotome, "Optimized Math Functions for a Fixed-Point DSP Architecture," in *19th Symposium on Computer Architecture and High Performance (SBAC-PAD)*, 2007, pp. 125–132.

[13] "CR-Libm, a library of correctly rounded elementary functions in double precision," http://lipforge.ens-lyon.fr/www/crlibm/.

[14] N. Shibata, "Efficient evaluation methods of elementary functions suitable for SIMD computation," *Computer Science - R&D*, vol. 25, no. 1-2, pp. 25–32, 2010.

[15] S. Chevillard, M. Joldes, and C. Lauter, "Sollya: an environment for the development of numerical codes," in *Proc. of the Third International Congress on Mathematical Software (ICMS'10)*. LNCS, Springer, 2010, pp. 28–31.

[16] G. Melquiond, "Gappa - génération automatique de preuves de propriétés arithmétiques," http://gappa.gforge.inria.fr/.

[17] C. Mouilleron and G. Revy, "Automatic Generation of Fast and Certified Code for Polynomial Evaluation," in *Proc. of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20)*. IEEE Computer Society, 2011, pp. 233–242.

[18] C. Bruel, "If-conversion SSA framework for partially predicated VLIW architectures." *Digest of the 4th Workshop on Optimizations for DSP and Embedded Systems*, 2006.

[19] B. D. de Dinechin, "From machine scheduling to VLIW instruction scheduling," *ST Journal of Research*, vol. 1, no. 2, 2004.

[20] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008.

[21] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[22] J.-M. Muller, "On the definition of ulp(x)," Laboratoire LIP, ENS de Lyon, Tech. Rep. 2005-09, 2005.

[23] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[24] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy, "Computing floating-point square roots via bivariate polynomial evaluation," *IEEE Trans. on Computers*, vol. 60, no. 2, pp. 214–227, 2011.

[25] D. Knuth, *Seminumerical Algorithms*, 3rd ed., ser. The Art of Computer Programming. Addison-Wesley, 1987, vol. 2.

[26] "OpenCL 1.2 Specification," version 15, released November 15, 2011, http://www.khronos.org/registry/cl/.