Accelerating Sine and Cosine Evaluation with Compiler Assistance

Peter Markstein
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94062, U.S.A.
peter.markstein@hp.com

Abstract

Some software libraries add special entry points to enable both the sine and cosine to be evaluated with one call for performance purposes. This paper proposes another method which does not involve new function names. By having the compiler front end recognize trigonometric function invocations, and replace them with a call to a common function which executes the code common to all the functions, followed by a short routine to produce the desired computation, it is possible to compute both the sine and cosine, when needed in about the same time as to compute only one of them.

1 Introduction

The routines to evaluate the sine and cosine functions have much code in common. If a program invokes both the sine and cosine routines for the same argument, most of the work in evaluating the first function is repeated when invoking the second. Some program libraries exploit this fact by providing alternative routines that return both $\sin x$ and $\cos x$ with one function call. A variant returns the complex quantity $\operatorname{cis} x = \cos x + i \sin x$. The drawback to these new functions is that they are not part of the standard programming interface, and codes that exploit them thereby become non-portable.

Some compilation systems recognize calls to the sine and cosine routines with the same argument, and replace them with a call to a "sincos" function that returns both values. This represents a new style of optimization, and it requires that the programmer know exactly the scope over which the transformation can apply.

In addition, the routines that return $\sin x$ and $\cos x$ concurrently may compute them in a different manner than the standard routines, altering their rounding behavior, accuracy or edge-case behavior, in the interest of speed. The search for means to accelerate the sine and cosine function arises because in practice, both functions are frequently called with the same argument. This is frequently the case in applications involving complex variables.

This paper discusses how a compiler can decompose invocations of sines and cosines into a pair of subroutine invocations, and then use existing compiler transformations and optimizations to achieve the same economies as the special purpose routines. The advantage of this approach is that it does not require an extended application program interface (API) to achieve performance improvements and it always uses the same algorithm. Therefore the characteristics of the sine and cosine routines as proposed here are not affected by whether they are evaluated one at a time or together. In addition, the compiler is not required to explicitly find instances where calls to sine and cosine are in near proximity. The technique can also cope with the tangent function, and it is readily adoptable to other function families which have much code in common, such as the hyperbolic functions, or the square root and reciprocal square root functions.

2 Method

Evaluation of the trigonometric functions involves several steps[1]:

1. reducing the argument to the first octant or smaller
2. extracting from a table, the sine and cosine of a value based on the argument reduction
3. computing the sine and cosine of the reduced argument by a Taylor, Chebychev, or Remez approximation
4. combining the table-lookups and approximation results to compute the desired function

The first three steps are common to the evaluation of sine and cosine. It is only how the results of these steps are combined that differentiates the sine and cosine functions. If the tangent is computed by dividing sine by cosine, then the tangent also fits into this paradigm.

To be specific, given an argument x, the first three steps generally lead to a reduced argument w, where $w = x - 2n\pi - t$, with n an integer and $|t| \leq \pi$. The variable t is chosen to make w smaller than a predetermined value, and t holds a value for which $\sin t$ and $\cos t$ can be obtained by table-lookup. Then $\sin x = \sin(w + t) = \sin w \cos t + \cos w \sin t$. Similarly, $\cos x = \cos w \cos t - \sin w \sin t$. Only $\sin w$ and $\cos w$ must be approximated by code, and both are needed for the evaluation of the sine and cosine functions. By constraining w to be sufficiently small, polynomials of low degree will suffice to compute $\sin w$ and $\cos w$ to the required precision. From the above discussion, the sine and cosine functions differ only in how these four values are combined in the last step.

The common steps can be exploited by introducing new functions into the repertoire of functions known by the compiler: trigstart, sinfinish, cosfinish, and perhaps tanfinish. The trigstart function performs the first 3 steps outlined above, and returns a structure, in the floating point registers, with the multiple computed values. The sinfinish, cosfinish and tanfinish routines take the structure produced by trigstart, and from it compute sin, cos, and tan respectively of the argument that had been given to trigstart. (On some platforms, the linkage conventions may not permit return of a structure in floating point registers. In such cases the values must be returned via storage, imposing a small overhead to retrieve these values. But since the return values would most likely be in a nearby cache, the penalty for streaming the structure back to registers should be no worse than one cache access).

The function trigstart could return a structure containing four values: $\sin t$, $\cos t$, $\sin w$, and $\cos w$. In practice, it has been useful to also return w, to allow the results of $\sin(\pm 0)$and $\tan(\pm 0)$ to retain the sign of the argument. It is also possible for trigstart to return $(\sin w)/w$ instead of $\sin w$, and $(\cos w - 1)/w$ instead of $\cos w$, in which case the code for the three finishing routines would be slightly different, but slightly more precise[3]. Computing $\sin w/w$ does not involve a division; it merely computes $(1 - w^2/3! + w^4/5! - ...)$, and similarly for the expression involving $\cos w$. So trigstart becomes two operations shorter to compensate for the extra instruction introduced into each finish routine. On a machine with fused multiply-add (FMA), the finishing routines each require 3 operations. For example, the sine function would be computed

$$\begin{aligned} \sin x &= \sin(w + t) = \sin t \cos w + \cos t \sin w \\ &= \sin t + w(\sin t \frac{\cos w - 1}{w} + \cos t \frac{\sin w}{w}) \end{aligned}$$

The alternate form allows the result to be expressed as a small adjustment to the table lookup value. For the parenthesized expression, it is better to compute the smaller product directly with a multiplication operation, and then the full expression with an FMA. Here, either product could be the smaller. But the first is bounded by w, and the second by 1.0, so our code computes the first product directly.

The front end of the compiler is modified to decompose the trigonometric functions as follows:

```
sin(x) = sinfinish(trigstart(x))
cos(x) = cosfinish(trigstart(x))
tan(x) = tanfinish(trigstart(x))
```

In the compiler's intermediate language, `trigstart` is treated like an ordinary arithmetic operation (a pure function), which always computes the same result for a given argument, without any side effects. During early stages of optimization, `trigstart` can be subjected to all the usual optimizations, such as common sub-expression elimination and code motion[4]. If a program contains the code: `a = sin(x); b = cos(x);` the compiler would generate:

```
R100 = trigstart(x)
A = sinfinish(R100)
R100 = trigstart(x)
B = cosfinish(R100)
```

and an optimization would remove the second computation of trigstart [1]. Even constructions such as:

```
if (x > y) z = sin(x);
else z = cos(x);
```

where the invocations of sine and cosine are in different control flow paths would result in compiled code which behaves as though it had been written as:

```
tmp = trigstart(x);
if (x > y) z = sinfinish(tmp);
else z = cosfinish(tmp);
```

The compiler achieves this transformation by code motion or code hoisting, to move the computation of trigstart(x) in front of the conditional branch.

The names trigstart, sinfinish, cosfinish, and tanfinish need never be made part of the API. In actual practice, the name given to trigstart would be outside the user's name space. The names of the finishing routines are used here only for expository purposes as the compiler is expected to expand these 3-operation functions inline (see the example in section 3).

2.1 Hyperbolic functions

As an example of another function family amenable to this treatment, the hyperbolic functions can be expressed in

[1] Some compilers may assign a different symbolic register to the second computation of trigstart, but would then know how to remove the second computation and alias its result to that of the first computation.

terms of $\mathrm{expm1}(x) = e^x - 1$ as follows:

$$\sinh x = \frac{1}{2}\,\mathrm{expm1}(x) + \frac{\mathrm{expm1}(x)}{2(\mathrm{expm1}(x)+1)}$$

$$\cosh x = \frac{[\mathrm{expm1}(x)]^2}{2(\mathrm{expm1}(x)+1)} + 1$$

A hyperbolic start function could return two values: $u = \mathrm{expm1}(x)$ and $v = \frac{1}{2} \cdot \mathrm{expm1}(x)/(\mathrm{expm1}(x)+1)$. Then one can compute $\sinh x = \frac{1}{2}u + v$, and $\cosh x = uv + 1$.

An alternate method of computing the hyperbolic functions [3] allows a treatment similar to that shown for the trigonometric functions. Decompose the argument x as $x = n \ln 2 + t$, with n an integer and $|t| < \frac{1}{2} \ln 2$. The hyperbolic start function returns a structure with 4 or 5 values similar to those returned by `trigstart`. These values are combined according to the hyperbolic function addition formulas in the finishing routines to obtain the desired results. Within the start routine, $\sinh(n \ln 2)$ is computed simply as $(2^n - 2^{-n})/2$, and $\cosh(n \ln 2)$ as $(2^n + 2^{-n})/2$.

3 Results

The HP-UX compiler for Itanium has implemented the methodology described above for trigonometric functions by modification of the front ends of the C/C++ and Fortran compilers to recognize invocations of the sine, cosine, and tangent functions. The optimizations to remove and/or reposition redundant trigstart operations occur without any change to the compiler's optimizer, since trigstart is regarded as a primitive operation during early rounds of optimization. In C, the user must include a reference to the header `math.h` to enable the compiler to decompose the trigonometric functions as described above. The code

```
s = sin(x);
c = cos(x);
```

results in the compiler-generated assembly code shown in Figure 1 (comments inserted by the author).

Instructions between double semicolons execute concurrently in Itanium systems with two floating point units, allowing the sine and cosine to finish in the same time together as either one would alone. Instructions with the "`.s1`" completer do not modify the user-visible floating point status register, and they are executed in double-extended precision. Floating point instructions with the "`.d.s0`" completer may affect the floating point status register, and their results are rounded to double precision.

The HP-UX Itanium compiler at high optimization (+O3) also allows complete inlining of the trig functions, in which case the trigstart function would be expanded in the calling routine. When this occurs in a loop, code motion will move the invariant instructions to the point just before loop entry. Thus all the constants used by circular functions would be loaded into registers before the start of the loop, and only accessed from registers during execution of the loop. In a loop such as:

```
//argument x in fp reg. 8
//the instruction below is Itanium's call instruction
br.call.sptk.few rp = trigstart;;
//on return, f8 = cos t; f9 = (cos w - 1)/w; f10 = sin t;
// f11 = sin w/w; f12 = w.
//Note: fma a=b,c,d computes a=d+b*c;
// fnma a=b,c,d computes a=d-b*c
fmpy.s1 f6 = f10, f9          // f6 = sin t * (cos w - 1.0) / w
fmpy.s1 f7 = f8, f9 ;;        // f7 = cos t * (cos w - 1.0) / w
fma.s1 f6 = f8, f11, f6       // f6 = f6 + cos t * (sin w) / w
fnma.s1 f7 = f10, f11, f7;; // f7 = f7 - sin t * (sin w) / w
fma.d.s0 f6 = f12, f6, f10  //  s = sin(x) = sin t + w * f6
fma.d.s0 f8 = f12, f7, f8;;  // c = cos(x) = cos t + w * f7
```

Figure 1. Compiler-Generated Assembly Language Code for Computing Sin(x) and Cos(x)

```
for (i = 0; i < n; i++) {
   s[i] = sin(x[i]);
   c[i] = cos(x[i]);
}
```

the invocation of both sine and cosine requires only three floating point operations beyond only invoking either one. The loop will be software pipelined, giving performance associated with vector machines, as shown in Table 1.

Of course, there is only one way to compute these trig functions; no shortcuts are taken during trigstart, whether or not it is inlined. HP-UX's trigstart implements the careful Payne-Hanek [5] argument reduction, but it is clever enough to only perform it when needed[3]. On the average, it is invoked once in 16000 evaluations. It adds about 40 cycles, when needed, to the evaluation of trigstart, or to a standard closed sine or cosine function.

Using C99 [6] the function $\mathrm{cis}\, x = \cos x + i \sin x$ can be realized efficiently with a tiny program.

```
#include <math.h>
#include <complex.h>
//In C99 complex arithmetic
//I is the imaginary unit.
double complex cis(double x) {
   return cos(x) + I * sin(x);
}
```

With compiler assistance as proposed here, optimal assembly code is produced, similar to the code shown in Figure 1.

Table 1 shows the timing of the double precision sine and cosine functions, when invoked in isolation or together. For

	Closed Routine	Software Pipelined
$\sin x$	53	7.37
$\cos x$	53	7.37
$\sin x$ and $\cos x$ (compiler assist)	53	9.16
$\sin x$ and $\cos x$ (separate calls)	106	15.37

Table 1. Timings for Itanium II (in machine cycles)

computing both functions, timings are given for the method described in this paper, as well as for computation by two function calls. The software pipelined loops were not unrolled.

4 Conclusion

By decomposing each trigonometric function into two routines:

a routine with operations common to all the functions, and

a short finishing routine that combines the results of the common routine,

it is now possible to invoke several trigonometric functions of the same argument without introducing a new API for special functions that compute them with one invocation. The advantage is that the standard function names are the only names that need be used. Standard compiler optimization removes redundant computations of the code common to trigonometric functions of the same argument.

Whether the trigonometric functions are evaluated by closed routines or inlined into what will become a vectorized loop, only the standard function names need be referenced. If more than one trigonometric function is evaluated for the same argument, only one instance of the $\mathrm{trigstart}$ evaluation is performed, and in no case are the precision or corner case behavior of the routines compromised for performance. The desired effect of computing both $\sin x$ and $\cos x$ together in the same time as it takes to compute one of them is accomplished without introducing new special functions into the library.

Two similar approaches pertain to the hyperbolic functions. Square root and reciprocal square root also respond to a similar treatment. This approach is particularly advantageous for hardware which lacks a square root instruction; on such machines the square root is typically computed in terms of the reciprocal square root.

5 Acknowledgements

Realizing the method described here in HP-UX was the result of close collaboration of many people. Jim Thomas led the effort, and was supported by Ren-Cang Li and Jon Okada from his Math Library project[2]. On the compiler side, Kevin Crozier, Theresa Johnson and David Gross designed and implemented the required compiler modifications.

References

[1] S. Gal, B. Bachelis, *An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard*, ACM Transactions on Mathematical Software 17, 1 (March 1991), 26-45

[2] Ren-Cang Li, Peter Markstein, Jon P. Okada, James W. Thomas, *The Libm Library and Floating-Point Arithmetic in HP-UX for Itanium 2*, 2002, http://h21007.www2.hp.com/dspp/files/unprotected /Itanium/FP_White_Paper_v2.pdf

[3] Peter Markstein, *IA-64 and Elementary Functions: Speed and Precision*, Prentice-Hall, New Jersey, 2000 .

[4] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufman, San Francisco, CA 1997.

[5] Jean-Michel Muller, *Elementary Functions - Algorithms and Implementation*, Birkhauser, Boston·Basel·Berlin, 1997.

[6] *Programming Languages – C*, International Standard ISO/IEC9899:1999(E), 1999.