A Fast-Start Method for Computing the Inverse Tangent

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

Abstract

In a search for an algorithm to compute atan(x) which has both low latency and few floating point instructions, an interesting variant of familiar trigonometry formulas was discovered that allow the start of argument reduction to commence before any references to tables stored in memory are needed. Low latency makes the method suitable for a closed subroutine, and few floating point operations make the method advantageous for a software-pipelined implementation.

1 Introduction

Short latency and high throughput are two conflicting goals in transcendental function evaluation algorithm design. In a system that allows function inlining and software pipelining at high optimization, and closed subroutine calls at low optimization (or debugging), the question arises whether the software pipelined algorithm must agree for all arguments with the closed subroutine's algorithm. Lacking a solution to the Table Maker's Dilemma [6], and the undesirableness of different results at different performance levels, we searched for algorithms having both good latency characteristics and good throughput behavior. Some compromise in both the closed subroutine performance and vectorized performance will be the consequence of requiring the identical results in both implementations.

The optimal technique for high throughput is to use as few instructions (especially floating point instructions) as possible, as the number of floating point instructions in an algorithm divided by the number of floating point operations that can be dispatched in a cycle gives the limiting software pipelined cost for an evaluation. (It is possible to break out of a software pipelined loop to handle rare special cases. In trigonometric algorithms, for example, the use of the Payne-Hanek argument reduction method is handled out-of-loop because its use is rarely required.)

For short latency, it is often desirable to use many instructions. Different sequences may be optimal over various subdomains of the function, and in each subdomain, instruction level parallelism, such as Estrin's method of polynomial evaluation [3] reduces latency at the cost of additional intermediate calculations.

In searching for an inverse tangent algorithm, an unusual method was discovered that permits a floating point argument reduction computation to begin at the first cycle, without first examining a table of values. While a table is used, the table access is overlapped with most of the argument reduction, helping to contain latency. The table is designed to allow a low degree odd polynomial to complete the approximation. A new approach to the logarithm routine [1] which motivated this study also begins argument reduction immediately to reduce latency.

In Section 2 some previous approaches to inverse tangent approximation are described. Section 3 describes the new algorithm, and Section 4 compares the classic and new algorithms.

2 Prior Art

One of the lowest latency inverse tangent routines is described in [1]. For arguments x no larger than 1 in magnitude, a 47th degree odd polynomial (Chebychev or Remez approximation to arctan) is evaluated. This is essentially a 23rd degree polynomial in x^2. Exploiting the fact that modern processors have fully pipelined floating point units, the polynomial is evaluated by Estrin's method by first evaluating terms of the form $t_{4n+3} + t_{4n+5}x^2$, e.g. $t_{47,45} = t_{47} + t_{45}x^2$. The subscripts of the results indicate the coefficients that are incorporated in the term. Each of these computations is independent of the others. On a machine that requires four cycles to complete one floating point instruction but

can issue two instructions concurrently, 8 of these operations can be dispatched by the time the first one finishes. These terms are then combined in parallel by multiplying by powers of x^4 (e.g. $t_{47...41} = t_{47,45}x^4 + t_{43,41}$). There are 5 extra instructions to evaluate higher powers of x, and a total of 29 instructions are required for the evaluation. These can be dispatched two per cycle, until near the end, when the dependencies between the final additions cannot be hidden. A similar strategy is used for arguments larger than 1 in magnitude, except that the polynomial is in powers of $1/x$. To avoid a division before the polynomial evaluation, parallelism is again exploited by writing the approximation as a 45th degree odd polynomial divided by x^{45}. The computation of x^{-45} proceeds in parallel with the polynomial evaluation and becomes essentially hidden.

This technique is not appropriate for software pipelining because it contains an excessive number of floating point computations. Even though not all computations are executed for each argument, in a software pipelined structure, it costs time to skip over the unneeded instructions.

Another low latency technique, for arguments less than 1 in magnitude, is to divide the domain [0,1] into n equal segments. For each segment i, the arctangent of the midpoint of the interval is tabulated, as well as polynomial coefficients to approximate

$$\arctan x = p_i + \sum_{j=1}^{k} c_{i,j}(x - p_i)^j \qquad (1)$$

where k depends on the number of segments n and the desired precision. Because the approximation at the i^{th} point is not centered at zero (except when i is zero), the approximating polynomial contains all terms. Gal et. al. [2] have proposed 256 segments, which require a (different) fifth degree polynomial for each segment. For arguments greater than 1 in magnitude, $1/x$ must first be computed, before the same technique is applied to the second octant. As in the previous example, the computation of $1/x$ can be overlapped with the table lookup to find p_i, and the coefficients $c_{i,j}$. Latency may be slightly shorter for results in the first octant. To reduce table size, we had divided the first quadrant into 14 segments, and used a 15^{th}degree polynomial with Estrin's method to approximate the arctan of the reduced argument.

A method used by Story and Tang at Intel [7] to compute atan(x/y) can be adopted for the simpler inverse tangent computation, by setting $y = 1$. For $1/8 \leq |x| \leq 1$, they propose deriving from $x = 2^k\,(1.x_{1,}x_{2,}x_{3,}x_{4,}x_{5,}x_{6,}...)$ a quantity

$$B = 2^k(1.x_{1,}x_{2,}x_{3,}x_{4,}x_{5,}1, 0, ...0) \qquad (2)$$

Then,

$$\arctan x = \arctan B + \arctan \frac{x - B}{1 + xB} \qquad (3)$$

Even before $\arctan B$ is found by table lookup based on k and the bits $x_{1,}x_{2,}x_{3,}x_{4,}x_5$, evaluation of the fraction in the Equation 3 can begin. However, the table construction method does not carry over to cases when $|x| \geq 1$, in which case an approximation to $1/x$ would first have to be generated, and B computed from that reciprocal approximation. The final result in that case exploits the identity

$$\arctan x + \arctan 1/x = \pi/2, \text{ for } x \geq 0$$

This method has the advantage over Gal's method in being able to start computation before table lookup, but in the event that the argument is larger than 1 in magnitude it requires a delay for the reciprocal approximation as well as the evaluation of a slightly different formula than the one given in Equation 3. This may increase the total number of floating point operations in the algorithm, In any case, there is some delay in deriving B from x. To date, this method has only been used for double-extended evaluations of the $\arctan$ function.

For high throughput evaluation of $\arctan(x)$, the first quadrant is partitioned into n equal sectors. By comparing the argument against the tangents of each sector boundary, one can quickly determine in which sector the result will lie. For each sector i, a table contains the midpoint p_i of the sector, as well as $\sin p_i$ and $\cos p_i$. If $y = \arctan(x)$, the object is to compute $g = \tan(y - p_i)$. Once g is known, $\arctan g$ is computed with a short odd polynomial, and the result is $p_i + \arctan(g)$. The quantity g is derived from the standard trigonometry formula by

$$\begin{aligned} g &= \frac{\tan y - \tan p_i}{1 + \tan y \tan p_i} \\ &= \frac{\cos p_i \tan y - \sin p_i}{\cos p_i + \tan y \sin p_i} \\ &= \frac{x \cos p_i - \sin p_i}{\cos p_i + x \sin p_i} \end{aligned} \qquad (4)$$

The advantage of the computation in Equation 4 is the avoidance of infinities in the case when $p_i = \pi/2$. The count of floating point instructions is competitive with Gal's method with a substantially smaller table because the approximating polynomial is odd, but from a latency viewpoint, the drawback is that the division cannot be started until i is determined, the table lookups have been completed and the numerator and denominator have been computed. We have found that dividing the first quadrant into 12 sectors allows a 11th degree approximation to yield over 64 bits of precision.

3 New Algorithm

The method we employ is similar to the last method described in Section 2. Given an argument x, we seek to find y such that $\tan y = x$. The first quadrant is divided into n sectors (with the i^{th} sector centered at $\mathrm{p}_i = \frac{\pi i}{2n}, i = 0, 1, ..., n$, and the segment boundaries at $b_i = \frac{\pi(2i-1)}{4n}, i = 1, 2, ..., n$.) For each sector, b_i, p_i, $\sin p_i$ and $\cos\ p_i$ are tabulated. By comparing x to the b_i, i is determined such that $\tan p_i$ is closest to y. So far, we have followed the previous scheme.

From a latency viewpoint, it would be desirable to avoid the division by a quantity that depends on i and the values of $\sin p_i$ and $\cos\ p_i$ (which are obtained by table lookup). Instead of computing $g = \tan(y - p_i)$, consider computing $h = \sin(y - p_i)$. If h can be obtained independently of the table lookup, then the computation of h can overlap the table lookups, hiding the latency of the previous method.

$$\begin{aligned} h &= \sin(y - p_i) \\ &= \sin y \cos p_i - \cos y \sin p_i \\ &= \tan y \cos y \cos p_i - \cos y \sin p_i \\ &= \cos y(\tan y \cos p_i - \sin p_i) \\ &= \frac{x \cos p_i - \sin p_i}{\sqrt{1 + x^2}} \end{aligned} \tag{5}$$

In the computation shown in Equation 5, the numerator does depend on the table lookup values, but the more time-consuming computation of the reciprocal square root is strictly a function of the input. On a computer such as Itanium, which has a fused multiply-add instruction, as well as a reciprocal square root approximation instruction, the radicand can be computed with one instruction issued in the very first cycle of the implementation, and the inverse square root can be computed while the table lookup takes place. Once the table lookup is complete, a fused multiply-add computes the numerator in Equation 5, followed by a multiplication of the approximate reciprocal square root to give the reduced argument for an inverse sine approximation. For small arguments, the inverse sin approximations converge slightly faster than the inverse tangent, so that often, an approximating polynomial with one fewer term keeps the error below the desired bound. The inverse tangent is computed as

$$\arctan(x) = p_i + \arcsin\left(\frac{x \cos p_i - \sin p_i}{\sqrt{1 + x^2}}\right) \tag{6}$$

Compared to using Equation 1, this method has about the same number of floating point instructions, but there is no distinction made between results in the first and second octant and the bulk of the computation of the reduced argument always overlaps the table lookup process.

The method can easily be adopted to compute $\mathrm{atan2}(x, y) = \arctan(x/y)$ in the following manner:

$$\begin{aligned} \arctan x/y &= \mathrm{atan2}(x, y) \\ &= p_i + \arcsin\left(\frac{\frac{x}{y}\cos p_i - \sin p_i}{\sqrt{1 + \left(\frac{x}{y}\right)^2}}\right) \\ &= p_i + \arcsin\frac{x \cos p_i - y \sin p_i}{\sqrt{x^2 + y^2}} \end{aligned}$$

It takes two floating point operations to evaluate the radicand, and it will take two floating point operations to compute the numerator. It is also necessary to approximate $x/\sqrt{x^2 + y^2}$ in order to determine in which segment the result lies; however, an eight bit approximation reciprocal square root of the radicand is adequate. Evaluating $\mathrm{atan2}$ takes an additional three floating point operations when compared to the $\arctan$ evaluation.

Table 1 gives both the scalar latency and number of integer and floating point operations used on Itanium II for various subcomputations that may appear in the arctangent routines that we have discussed. In this way, the reader can experiment with the consequences of various methods of evaluating the arctangent. Itanium II can issue two memory access instructions per cycle. So if several consecutive memory accesses are issued, only the first appears to suffer the latency. Thus n consecutive floating point words can be accessed in $\lceil n/2 \rceil + 5$ cycles. Other integer and floating point operations may be issued while waiting for a load operation to complete. In each cycle, Itanium II can issue as many as two floating point operations, and six operations in all. For the vectorized version, if the number of floating point operations exceeds half of the number of integer operations, the average latency of a vectorized evaluation can be as short as 1/2 the number of floating point operations.

3.1 Error Analysis

If the computation of double precision arctan is carried out using double-extended arithmetic, and the tables are stored as double-extended precision quantities, then the error in using Equation 6 falls into four parts: the errors in computing $x \cos p_i - \sin p_i$, $1/\sqrt{1 + x^2}$, the error in the polynomial that approximates $\arcsin$, as well as rounding errors throughout the computation. The errors in $\cos p_i$, and $\sin p_i$ are bounded by $\frac{1}{2}$ of a double-

Operation	Scalar Latency	FP Ops	Integer or Memory Ops
move data between fpr and gpr	4-5		1
$1/x$, 34 bits of precision	16	5	
$1/x$, 62 bits of precision	20	7	
$1/\sqrt{x}$, 34 bits of precision	20	6	2
$1/\sqrt{x}$, 62 bits of precision	28	10	2
$1/\sqrt{x}$, 8 bit approximation	4	1	
Memory to fp register	6		1
Memory to gp register	2		1
Evaluate polynomial, k^{th} degree	$4k$	k	
Which of n sectors contains argument	$n+3$		$3n$
Find sector by binary search	$4\lceil \log_2 n \rceil$		$5\lceil \log_2 n \rceil$
fp add, multiply	4	1	
fp fused multiply-add	4	1	
integer add, shift, and, load immediate	1		1

Table 1. Latency and number of floating point operations of common computations on Itanium II. Since Itanium can issue two floating point operations per cycle, the vector latency is half the number of floating point operations.

extended precision unit in the last place[1]. This could be made even smaller by employing Gal's exact table method[2]. The computation of $x \cos p_i - \sin p_i$ will involve cancellation (since p_i was chosen so that x is close to $\tan p_i$.) The error in this computation is bounded by 1 ulp of the value of $\sin p_i$ (except when $i = 0$, in which case $x \cos p_i - \sin p_i$ is exactly x.) If Gal's exact table method is used, the error committed by computing $x \cos p_i \ominus \sin p_i$ will be bounded by $\frac{1}{2}$ ulp.

The error in computing $1 + x^2$ may be as large as $\frac{1}{2}$ ulp, so that value of $1/\sqrt{1 \oplus x^2}$may differ from the true value by as much as $\frac{1}{4}$ ulp. But computing $1/\sqrt{1 \oplus x^2}$ to double-extended precision using Goldschmidt's inverse square root algorithm (for low latency and few operations) may introduce an error of as much as 2 ulps of the computational precision [5]. Multiplying the inverse square root approximation by $x \cos p_i \ominus \sin p_i$ may add another $\frac{1}{2}$ ulp error, so that the error in the reduced argument is bounded by $2\frac{3}{4}$ ulps, plus an ulp of p_i whenever $i \neq 0$. With the exact table method, the error is bounded by $3\frac{1}{4}$ ulp.

When $i = 0$, the computation reduces to $\arctan(x) = \arcsin\left(x/\sqrt{1+x^2}\right)$. We have already seen that in this case, argument reduction may contribute an error as large as $2\frac{3}{4}$ ulps. The operations in evaluating the polynomial approximation to arcsin will introduce slightly over $\frac{1}{2}$ ulp, and the approximation itself (without regard to the arithmetic operations) may introduce another ulp. (One can reduce this somewhat, at the expense of additional instructions and latency or larger tables). Thus, the error in using this method for the sector containing 0 is bounded by $4\frac{1}{4}$ computational ulps. But since computational precision is 64 bits wide, and we seek a double precision result, the total error, in double precision ulps, is bounded by .5021 ulps.

When $i = 1$, and a negative reduced argument, the final addition in Equation 6 is effectively a subtraction, and the subtrahend may be as large as $\frac{1}{2}p_i$. Errors as large as $4\frac{1}{4}$ computational ulps may occur before rounding to single or double precision.

When $i > 1$, the method is more precise. The correction to p_i is less than $p_i/4$, so that the final addition in Equation 6 will introduce $\frac{1}{2}$ ulp, as well as the errors in argument reduction and the polynomial approximation.

Suppose that the polynomial is of sufficient degree to make the error in the approximation of $\arcsin(x)$ less than $\frac{1}{2}$ ulp in double extended precision (see [4] for tables of trade-offs between polynomial degree and n, the number of sectors in the first quadrant.) The total error of the computation is bounded by $4\frac{1}{4}$ double-extended ulps, or .0021 ulps in double precision. As i increases, the error contributed by the $\arcsin$ computation decreases, since the p_i increase while the magnitude of the $\arcsin$ term is bounded by $\sin \pi/4n$.

[1] In the discussion in Section 3.1, a unit in the last place, or ulp, is always a double-extended precision ulp, unless stated otherwise.

	Single Precision		Double Precision	
	Scalar	Vector	Scalar	Vector
New method	53	8.0	63	11.0
Eqn. 4	91	8.5	99	10.5
Eqn. 1	43	14	52	20
Intel's method [1]	40		56	

Table 2. Scalar and Vector performance of various arctan routines. Times are given in cycles. Scalar times refer to closed calls for a single argument. Vector times indicate the average time if a software-pipelined implementation is applied to a vector of arguments. Low numbers are better in all cases. Entries which are not known or not applicable are left blank.

4 Results

Implementing this method for Itanium, Table 2 shows the latency and throughput results for the new algorithm, as well as some of the algorithms described in Sec. 2. The scalar column gives the latency in cycles for a typical evaluation. The vector column gives how frequently (in cycles) a new arctan evaluation may start. For some of these algorithms, the vector performance has not been measured or evaluated. Of course, if it were not required for the scalar and vector results to be identical, then two algorithms could be used: a short latency algorithm such as described in [1], and a high throughput algorithm, as described in the paragraph containing Equation 4.

We have also sketched code for the function $\mathrm{atan2}$, and have measured the double precision vector performance at 15 cycles. However, by reading the code, we believe that better scheduling should eventually reduce the latency to 13.5 cycles.

5 Conclusion

The novelty suggested by this paper is to transform the computation of the inverse tangent to the computation of the inverse sine. By exploiting Equation 6 to compute the inverse tangent in terms of the inverse sine of the reduced argument, the argument reduction can begin immediately, even before it is determined in which segment the result will lie. This permits a relatively short latency for the closed function, while requiring few floating point instructions. Consequently the throughput of the algorithm is competitive with other vectorizable algorithms.

The algorithm for the inverse tangent shown in this paper emphasizes two techniques. The first is to find algorithms which allow computation on the argument(s) immediately, in parallel with the table lookup activity. The Intel development for the logarithm [1] was an inspiration to find such algorithms. The second point is that the function evaluated on the reduced argument need not be the same function as the one we are trying to approximate. We have shown the straightforward extension of this technique to the $\mathrm{atan2}$ function.

An advantage of the new method vis-a-vis use of Equation 1 is that the same polynomial is evaluated for each argument. Thus only one set of polynomial coefficients need be kept in registers while the vectorized algorithm executes. Had our method required a different polynomial for each argument, then the vectorized routine would be hindered by the requirement to keep many of the coefficients of the polynomials under evaluation in registers at the same time. The number of available registers may be insufficient for this purpose and may cause the seemingly short program (from the viewpoint of number of floating point instructions) to run slower than expected.

In comparison with the method of Story and Tang, the computation of B (see Equation 2) does take the equivalent of two or three floating point operations, so that the start of the division must wait about 3 floating point latencies to commence. On the other hand, division requires fewer floating point operations than reciprocal square root.

Acknowledgements

Jim Thomas and Jon Okada, colleagues at Hewlett-Packard, have contributed valuable discussions which helped with reducing these ideas into practice. They played key roles in the testing and refinement of the programs implementing this new method.

Several referees made valuable suggestions, leading to improvements in this paper.

References

[1] Marius Cornea, John Harrison, Ping Tak Peter Tang, *Scientific Computing on Itanium-based Systems*, Intel Press, 2002.

[2] 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

[3] Donald Knuth, *The Art of Computer Programming*, Vol II: *Seminumerical Algorithms*, Addison-Wesley, Reading, Mass, 1968.

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

[5] Peter Markstein, Software Division and Square Root Using Goldschmidt's Algorithms, *Proceedings of the 6th Conference on Real Numbers and Computers*, pp. 146-157, 2004.

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

[7] Shane Story, Ping Tak Peter Tang, New Algorithms for Improved Transcendental Functions on IA-64, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pp 4-11, 1999