Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

SPI

On-the-fly Range Reduction

Vincent Lefèvre, Jean-Michel Muller

Novembre 2000

Research Report N° 2000-34

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

INRIA

On-the-fly Range Reduction

Vincent Lefèvre, Jean-Michel Muller

Novembre 2000

Abstract

In several cases, the input argument of an elementary function evaluation is given bit-serially, most significant bit first. We suggest a solution for performing the first step of the evaluation (namely, the range reduction) *on the fly*: the computation is overlapped with the reception of the input bits. This algorithm can be used for the trigonometric functions $\sin$, $\cos$, $\tan$ as well as for the exponential function.

Keywords: Range reduction, Elementary functions, Computer arithmetic.

Résumé

Il arrive que l'oprande dont on doit calculer une fonction lmentaire soit disponible chiffre aprs chiffre, en srie, en commenant par les poids forts. Nous proposons une solution permettant d'effectuer la premire phase de l'valuation (la rduction d'argument) *au vol*: le calcul et la rception des chiffres d'entre se recouvrent. Cet algorithme peut tre utilis pour les fonctions trigonomtriques $\sin$, $\cos$, $\tan$, ainsi que pour l'exponentielle.

Mots-clés: Rduction d'argument, fonctions lmentaires, arithmtique des ordinateurs.

1 Introduction

The algorithms used for evaluating the elementary functions only give a correct result if the argument is within some bounded interval. To evaluate an elementary function $f(x)$ (sine, cosine, exponential,...) for any x, one must find some "transformation" that makes it possible to deduce $f(x)$ from some value $g(y)$, where

- y, called the *reduced argument*, is deduced from x;
- y belongs to the convergence domain of the algorithm implemented for the evaluation of g.

With the usual functions, the only cases for which reduction is not straightforward are the cases where y is equal to $x - nC$, where n is an integer and C a constant (for instance, for the trigonometric functions, C is a multiple of $\pi/8$).

Example 1 (Computation of the cosine function) *Assume that we want to evaluate* $\cos(x)$*, and that the convergence domain of the algorithm used to evaluate the sine and cosine of the reduced argument contains* $[0, +\pi/4]$*. We choose* $C = \pi/4$*, and the computation of* $\cos(x)$ *is decomposed in three steps:*

- *compute* y *and* n *such that* $y \in [0, +\pi/4]$ *and* $y = x - n\pi/4$*;*
- *compute* $g(y, n) =$

$$\left\{ \begin{array}{rll} \cos(y) & \textit{if} & n \textit{ mod } 8 = 0 \\ \frac{\sqrt{2}}{2}(\cos(y) - \sin(y)) & \textit{if} & n \textit{ mod } 8 = 1 \\ -\sin(y) & \textit{if} & n \textit{ mod } 8 = 2 \\ -\frac{\sqrt{2}}{2}(\cos(y) + \sin(y)) & \textit{if} & n \textit{ mod } 8 = 3 \\ -\cos(y) & \textit{if} & n \textit{ mod } 8 = 4 \\ \frac{\sqrt{2}}{2}(-\cos(y) + \sin(y)) & \textit{if} & n \textit{ mod } 8 = 5 \\ \sin(y) & \textit{if} & n \textit{ mod } 8 = 6 \\ \frac{\sqrt{2}}{2}(\cos(y) + \sin(y)) & \textit{if} & n \textit{ mod } 8 = 7 \end{array} \right. \tag{1}$$

- *obtain* $\cos(x) = g(y, n)$*.*

Example 2 (Computation of the exponential function) *Assume that we want to evaluate* e^x *in a radix-2 number system, and that the convergence domain of the algorithm used to evaluate the exponential of the reduced argument contains* $[0, \ln(2)]$*. We can choose* $C = \ln(2)$*, and the computation of* e^x *is then decomposed in three steps:*

- *compute* $y \in [0, \ln(2)]$ *and* n *such that* $y = x - n\ln(2)$*;*
- *compute* $g(y) = e^y$*;*
- *compute* $e^x = 2^n g(y)$*.*

Unless multiple-precision arithmetic is used during the intermediate calculations, a straightforward computation of y as $x - nC$ is to be avoided, since this operation will lead to catastrophic cancellations (i.e., to very inaccurate estimates of y) when x is large or close to an integer multiple of C. Many algorithms have been suggested for performing the range reduction accurately [1, 2, 3, 9, 11].

Now, there are many cases (on special-purpose systems) where the input argument of a calculation is generated most significant digit first. This happens, for instance, when this argument is the result of a division or a square root obtained through a digit-recurrence algorithm [7, 10], the output of an on-line algorithm [5, 12], or when it is generated by an analog-to-digital converter.

In the sequel of this paper, we present an adaptation of the Modular Range Reduction Algorithm [3, 8] that accepts such digit serial inputs and performs the range reduction "on the fly": most of the computation is overlapped with the reception of the input bits, and the reduced argument is produced almost immediately after reception of the last input bit. On-the-fly arithmetic algorithms have already been proposed by Ercegovac and Lang for rounding or converting a number from redundant to non-redundant representation [4, 6].

2 Notations

In the sequel of the paper, $x = x_h x_{h-1} \cdots x_0.x_{-1}x_{-2} \cdots x_\ell$ is the input argument, $C = 0.C_{-1}C_{-2} \cdots C_{-p}$ is the constant of the range reduction (with $-p \leq \ell$), and $y = 0.y_{-1}y_{-2} \cdots y_{-p}$ is the reduced argument. We assume $1/2 \leq C < 1$. These values satisfy:

- $0 \leq y < C$;
- $n = (x - y)/C$ is an integer.

We also define, for each i, m_i (also called $2^i \bmod C$) as the unique value between 0 and C such that $(2^i - m_i)/C$ is an integer. These notations give some contraints on x and C (e.g., C is less than 1, x is less than 2^{h+1}). One can easily adapt the algorithms given in the sequel of the paper to variables belonging to other domains. We chose these contraints to make the presentation of the algorithms simpler.

3 Non-redundant algorithm

Algorithm 1 is by far less efficient than the "redundant" algorithm given afterwards. We give it because it is simpler to understand, and because the other algorithm is derived from it. The basic idea is the following: at step i of the algorithm, when we receive input bit x_{h-i} of x, we add $x_{h-i} \times \left(2^i \bmod C\right)$ to an accumulator. If the accumulated value becomes larger than C, we subtract C from it.

Let us call A_{i+1} the value obtained after this operation. One can easily check that $0 \leq A_{i+1} < C$ and $A_{i+1} - x_h x_{h-1} \cdots x_{h-i} \times 2^{h-i}$ is an integer multiple of C. Hence the final value stored in the accumulator is equal to the reduced argument y.

A possible variant consists in computing $U_i = A_i + x_{h-i}\,(m_{h-i} - C)$ in parallel with T_i, and then to choose A_{i+1} equal to U_i if $U_i \geq 0$, otherwise T_i.

Algorithm 1 Non-redundant algorithm.

$A_0 = 0$
for $i = 0$ **to** $h - \ell$ **do**
  $T_i = A_i + x_{h-i}\, m_{h-i}$
  if $T_i < C$ **then**
    $A_{i+1} = T_i$
  else
    $A_{i+1} = T_i - C$
$y = A_{h-\ell+1}$

4 Redundant algorithm

Now, to accelerate the reduction, we assume that we perform the accumulations with *carry-save* additions. The carry-save number system allows very fast, carry-free additions. On the other hand, its intrinsic redundancy makes comparisons somewhat more complex. The accumulator will store the values A_i in carry-save. In the previous algorithm, we needed "exact" comparisons between the A_i's and C. Having the A_i's stored in carry-save makes these "exact" comparisons difficult. Instead of that, we will perform comparisons based on the examination of the first three carry-save positions of A_i only. This will not allow to bound the A_i's by C. Nevertheless, we will show that the A_i's will be upper-bounded by $C + \frac{1}{2}$ (therefore by $\frac{3}{2}$), which will suffice for our purpose. We denote:

$$A_i = \Big((A_{i,0}^{(1)}, A_{i,0}^{(2)}); (A_{i,-1}^{(1)}, A_{i,-1}^{(2)}); (A_{i,-2}^{(1)}, A_{i,-2}^{(2)}); \ldots ; (A_{i,-p}^{(1)}, A_{i,-p}^{(2)}) \Big)$$

where $A_{i,j}^{(1)}$ and $A_{i,j}^{(2)}$ are in $\{0, 1\}$ and

$$A_i = \sum_{j=0}^{p} (A_{i,j}^{(1)} + A_{i,j}^{(2)}) \cdot 2^{-j}.$$

The variable T_i of the non-redundant algorithm is used again, and is also represented in carry-save form:

$$T_i = \Big((T_{i,0}^{(1)}, T_{i,0}^{(2)}); (T_{i,-1}^{(1)}, T_{i,-1}^{(2)}); (T_{i,-2}^{(1)}, T_{i,-2}^{(2)}); \ldots ; (T_{i,-p}^{(1)}, T_{i,-p}^{(2)}) \Big)$$

This gives algorithm 2.

In the loop, we do not want to waste time with a full comparison to know whether we need to subtract C from T_i or not. Thus we use a rough approximation $\widehat{T}_i$ to T_i based on the first three digits of T_i. Since

$$\Big((T_{i,-3}^{(1)}, T_{i,-3}^{(2)}); \ldots ; (T_{i,-p}^{(1)}, T_{i,-p}^{(2)}) \Big) \le 2 \cdot 2^{-3} + 2 \cdot 2^{-4} + \cdots + 2 \cdot 2^{-p} < \frac{1}{2},$$

we have:

$$\widehat{T}_i \le T_i < \widehat{T}_i + \frac{1}{2}$$

Algorithm 2 Redundant algorithm.

$A_0 = 0 + 1$
for $i = 0$ **to** $h - \ell$ **do**
  $T_i = A_i +_{\text{cs}} x_{h-i}\, m_{h-i}$
  $\widehat{T}_i = \left((T_{i,0}^{(1)}, T_{i,0}^{(2)}); (T_{i,-1}^{(1)}, T_{i,-1}^{(2)}); (T_{i,-2}^{(1)}, T_{i,-2}^{(2)})\right) - 1$
    converted to non-redundant binary using a 3-bit adder
  if $\widehat{T}_i < C$ **then**
    $A_{i+1} = T_i$
  else
    $A_{i+1} = T_i -_{\text{cs}} C$ (or $T_i +_{\text{cs}} (1 - C) - 1$)
$B = A_{h-\ell+1} +_{\text{cs}} (1 - C)$
Convert $A_{h-\ell+1}$ and B to non-redundant binary.
if $B < 2$ **then**
  $y = A_{h-\ell+1} - 1$
else
  $y = B - 2$

We want to ensure that A_i is always positive, that is, $T_i - C$ does not lead to a negative number. Then, the subtraction is performed only when $\widehat{T}_i \geq C$. In this case, $T_i - C \geq \widehat{T}_i - C \geq 0$.

Now, we want to find an upper bound on all the A_i's (and one on the T_i's). Suppose that for a given i, we have $A_i \leq M$. Thus $T_i \leq M + C$. If $\widehat{T}_i < C$, then $A_{i+1} = T_i < \widehat{T}_i + \frac{1}{2} < C + \frac{1}{2}$; otherwise, $A_{i+1} = T_i - C \leq M$. If we choose $M = C + \frac{1}{2}$, then $A_{i+1} \leq M$ in both cases. By induction, $A_i \leq C + \frac{1}{2}$ and $T_i \leq 2C + \frac{1}{2}$ for all i.

The final value of y is converted to non-redundant representation using a conventional (i.e., non-redundant) addition. Another, faster, solution is to convert it on-the-fly, during the second loop of the algorithm, using Ercegovac and Lang's on-the-fly algorithm [4, 6] for conversion from redundant to non-redundant representation.

5 An example: computation of $\cos(1010.111)$.

We choose $C = \pi/4 \approx 0.1100101$ ($p = 7$). Since $x = 1010.111$, we have $h = 3$ and $\ell = -3$).

The values of the m_i's are:
$$\begin{cases} m_3 &= 2^3 \mod \pi/4 &\approx 0.0010011 \\ m_2 &= 2^2 \mod \pi/4 &\approx 0.0001001 \\ m_1 &= 2^1 \mod \pi/4 &\approx 0.0110111 \\ m_0 &= 2^0 \mod \pi/4 &\approx 0.0011011 \\ m_{-1} &= 2^{-1} \mod \pi/4 &= 0.1 \\ m_{-2} &= 2^{-2} \mod \pi/4 &= 0.01 \\ m_{-3} &= 2^{-3} \mod \pi/4 &= 0.001 \end{cases}$$

The carry-save representations of the variables T_i and A_i generated by the redundant algorithm are

$x_3 = 1$	$T_0 = \begin{cases} 1.0010011 \\ 0.0000000 \end{cases}$	$0 < C$	$A_0 = \begin{cases} 1.0010011 \\ 0.0000000 \end{cases}$
$x_2 = 0$	$T_1 = \begin{cases} 1.0010011 \\ 0.0000000 \end{cases}$	$0 < C$	$A_1 = \begin{cases} 1.0010011 \\ 0.0000000 \end{cases}$
$x_1 = 1$	$T_2 = \begin{cases} 1.0100100 \\ 0.0100110 \end{cases}$	$0.1 < C$	$A_2 = \begin{cases} 1.0100100 \\ 0.0100110 \end{cases}$
$x_0 = 0$	$T_3 = \begin{cases} 1.0000010 \\ 0.1001000 \end{cases}$	$0.1 < C$	$A_3 = \begin{cases} 1.0000010 \\ 0.1001000 \end{cases}$
$x_{-1} = 1$	$T_4 = \begin{cases} 1.0001010 \\ 1.0000000 \end{cases}$	$1 \geq C$	$A_4 = \begin{cases} 1.0010001 \\ 0.0010100 \end{cases}$
$x_{-2} = 1$	$T_5 = \begin{cases} 1.0100101 \\ 0.0100000 \end{cases}$	$0.1 < C$	$A_5 = \begin{cases} 1.0100101 \\ 0.0100000 \end{cases}$
$x_{-3} = 1$	$T_6 = \begin{cases} 1.0010101 \\ 0.1000000 \end{cases}$	$0.1 < C$	$A_6 = \begin{cases} 1.0010101 \\ 0.1000000 \end{cases}$

We then get $y = 0.1010101$, whereas the exact value of $x \bmod \pi/4$ is $0.10101010001\ldots$.

6 Conclusion

The redundant algorithm presented in Section 4 allows fast, on-the-fly, range reduction. The accuracy of this method is the same as that of the Conventional Modular range reduction method (see [3, 8]).

References

[1] W. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[2] W. J. Cody. Implementation and testing of function software. In P. C. Messina and A. Murli, editors, *Problems and Methodologies in Mathematical Software Production*, Lecture Notes in Computer Science 142. Springer-Verlag, Berlin, 1982.

[3] M. Daumas, C. Mazenc, X. Merrheim, and J. M. Muller. Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions. *Journal of Universal Computer Science*, 1(3):162–175, March 1995.

[4] M. D. Ercegovac and T. Lang. On-the-fly conversion of redundant into conventional representations. *IEEE Transactions on Computers*, C-36(7), July 1987. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[5] M. D. Ercegovac and T. lang. On-line arithmetic: a design methodology and applications in digital signal processing. In *VLSI Signal Processing III*, pages 252–263, 1988. Reprinted

in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[6] M. D. Ercegovac and T. Lang. On-the-fly rounding. *IEEE Transactions on Computers*, 41(12):1497–1503, December 1992.

[7] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, 1994.

[8] J.M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[9] M. Payne and R. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18:19–24, 1983.

[10] J. E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-7:218–222, 1958. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[11] R. A. Smith. A continued-fraction analysis of trigonometric argument reduction. *IEEE Transactions on Computers*, 44(11):1348–1351, November 1995.

[12] K. S. Trivedi and M. D. Ercegovac. On-line algorithms for division and multiplication. In *3rd IEEE Symposium on Computer Arithmetic*, pages 161–167, Dallas, Texas, USA, 1975. IEEE Computer Society Press, Los Alamitos, CA.