IEICE Proceeding Series

Fast Multiprecision Algorithm like Quad-Double Arithmetic

Naoya Yamanaka, Shin'ichi Oishi

Vol. 2 pp. 433-436
Publication Date: 2014/03/18
Online ISSN: 2188-5079

2013 International Symposium on Nonlinear Theory and its Applications
NOLTA2013, Santa Fe, USA, September 8-11, 2013

Fast Multiprecision Algorithm like Quad-Double Arithmetic

Naoya Yamanaka† and Shin'ichi Oishi‡

†Research Institute for Science and Engineering, Waseda University
3-4-1 Okubo Shinjuku, Tokyo, 169-8555 Japan
‡Faculty of Science and Engineering, Waseda University
3-4-1 Okubo Shinjuku, Tokyo, 169-8555 Japan
Email: naoya_yamanaka@suou.waseda.jp, oishi@waseda.jp

Abstract—A quad-double number is an unevaluated sum of four double precision numbers, capable of representing at least 212 bits of significand. Hida *et. al.* have developed the *well-known* software for quad-double arithmetic called "QD". Similarly to their algorithms, in this paper, fast multiprecision algorithms are proposed. The proposed algorithms in this paper are designed to achieve the results as if computed in almost 4-fold working precision. Numerical results are presented showing the performance of the proposed multiprecision algorithms.

1. Introduction

In a numerical calculation sometimes we need higher than double precision floating point arithmetic to allow us to be confident a result. One alternative is to rewrite the program to use a software package implementing arbitrary-precision extended floating-point arithmetic such as MPFR [1] or exflib [2], and try to choose a suitable precision. There are possibilities intermediate between the largest hardware floating point format and the general arbitrary precision software which combine a considerable amount of extra precision with a relatively speaking modest factor loss in speed. An alternative approach is to store numbers in a multiple-component format, where a number is expressed as unevaluated sums of ordinary floating-point words, each with its own significand and exponent like ARPREC [3]. The multiple-digit approach can represent a much larger range of numbers, whereas the multiple-component approach has the advantage in speed.

We note that many applications would get full benefit from using merely a small multiple the working precision without the need for arbitrary precision. Bailey [4] have developed algorithms for double-double arithmetic[1]. A double-double number is an unevaluated sum of two double precision numbers, capable of representing at least 106 bits of significand. Similarly, Hida *et.al.* have implemented a quad-double arithmetic [5]: a quad-double number is an unevaluated sum of four double precision numbers, capable of representing at least 212 bits of significand. These algorithms can be made faster than some softwares for arbitrary precision.

[1] K. Briggs also have proposed the same idea.

In this paper we will propose an efficient format and describe fast algorithms for basic operations similar to quad-double arithmetic. Proposed format is also an unevaluated sum of four double precision numbers, but it is capable of representing at least 203 bits of significand (50 bit $\times 3 + 53$ bit). Hence, it is slightly less accuracy than a quad-double arithmetic, however, there is leeway to treat "carry bits" on IEEE 754 double precision numbers. As a result, presented algorithms for the format is faster than that for quad-double arithmetic. For example, proposed multiplication algorithm is about 2-times faster than that of quad-double arithmetic. By numerical experiments it is shown that the proposed algorithms are efficient compared with Hida *et.al*'s quad-double, MPFR and exflib.

2. Preliminary

2.1. Notation

We assume that neither overflow nor underflow occur. The set of floating-point numbers which have m-bit mantissa is denoted by $\mathbb{F}_m$. Simply, $\mathbb{F}$ denotes the set of floating-point numbers according to the IEEE 754 arithmetic standard. The relative rounding error unit, the distance from 1.0 to the next larger floating-point number in $\mathbb{F}_m$, is denoted by $\mathbf{u}_m$. For IEEE754 double precision $\mathbf{u} = 2^{-53}$. We denote by $fl(\cdot)$ the result of a floating-point operations in rounding to nearest corresponding to the IEEE 754 arithmetic standard.

Let us denote ufp ("unit in the first place") by

$$0 \neq r \in \mathbb{R} \quad \Rightarrow \quad \mathrm{ufp}(r) := 2^{\lfloor \log_2 |r| \rfloor}, \tag{1}$$

where we set $\mathrm{ufp}(0) := 0$ [6]. Furthermore, we define the floating-point predecessor and successor of a real number r by

$$\mathrm{pred}_m(r) := \max\{f \in \mathbb{F}_m \mid f < r\} \tag{2}$$
$$\mathrm{succ}_m(r) := \min\{f \in \mathbb{F}_m \mid r < f\}. \tag{3}$$

A floating-point number f is called a (m-bit) faithful rounding of a real number r if there is no other floating-point number between f and r [6]. It follows that $f = r$ in case $r \in \mathbb{F}_m$.

Definition 1
A floating-point number $f \in \mathbb{F}_m$ is called a faithful rounding of a real number $r \in \mathbb{R}$ if $pred_m(f) < r < succ_m(f)$. We denote this by $f \in faithful_m(r)$. For $r \in \mathbb{F}_m$ this implies $f = r$.

Rump *et.al.* have extended Definition 1 to a sequence $a_0, \cdots, a_{k-1}$ [7].

Definition 2
A sequence $a_0, \cdots, a_{k-1} \in \mathbb{F}_m$ is called a (m-bit and k-fold) faithful rounding of $s \in \mathbb{R}$ if

$$a_i \in faithful_m\left(s - \sum_{v=0}^{i-1} a_v\right) \quad for \quad 0 \leqq i \leqq k-1. \quad (4)$$

Furthermore, for the integer n, the numbers $a_0, \cdots, a_{k-2} \in \mathbb{F}_m$ and $a_{k-1} \in \mathbb{F}_n$, a sequence $a_0, \cdots, a_{k-1}$ is also called a (m-bit, n-bit and k-fold) faithful rounding of s if

$$a_i \in faithful_m\left(s - \sum_{v=0}^{i-1} a_v\right) \quad for \quad 0 \leqq i \leqq k-2, \quad (5)$$

$$a_{k-1} \in faithful_n\left(s - \sum_{v=0}^{k-2} a_v\right). \quad (6)$$

2.2. Error Free Transformations

It is known that the error of every floating-point operation is itself a floating-point number:

$$x = fl(a \circ b) \quad \Longrightarrow \quad x + y = a \circ b \quad \text{with} \quad y \in \mathbb{F} \quad (7)$$

for $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \cdot\}$. Remarkably, the error y can be calculated using only basic floating-point operations. First, we introduce the addition algorithm TwoSum. Knuth [8] presented the following algorithm which transforms a pair (x, y) with $x, y \in \mathbb{F}$ into a new pair (a, b) with $a, b \in \mathbb{F}$ satisfying

$$x + y = a + b \quad \text{with} \quad a = fl(x + y), \quad (8)$$

$$|b| \leq \mathbf{u}\,|a|. \quad (9)$$

Algorithm 1 Error-free transformation of the sum of two floating-point numbers.

```
function [a, b] = TwoSum(x, y)
    a = fl(x + y)
    c = fl(a − x)
    b = fl((x − (a − c)) + (y − c))
end
```

Next, we proceed to the dot product. The multiplication routine needs to split the input arguments into two parts. The following algorithm on **Algorithm 2** by Dekker [9] splits a floating point number $x \in \mathbb{F}$ into two parts $x_h, x_t \in \mathbb{F}$, where both parts have at most 26 nonzero bits, such that $x = x_h + x_t$.

Algorithm 2 Error-free transformation of the split a floating-point number into two floating-point numbers.

```
function [x_h, x_t] = Split(x)
    c = fl((2^27 + 1) · x)
    x_h = fl(c − (c − x))
    x_t = fl(x − x_h)
end
```

There is another splitting algorithm called "ExtractScalar". This algorithm is to extract high order part of a floating point numbers. A floating-point number is split relative to σ, a fixed power of 2. The higher and the lower part of the splitting may have between 0 and m significant bits, depending on σ.

Algorithm 3 Error-free transformation extracting high order part.

```
function [q, p'] = ExtractScalar(σ, p)
    q = fl((σ + p) − σ)
    p' = fl(p − q)
end
```

There, a 53-bit floating-point number is split into two parts relative to its exponent, and using a sign bit both the high and the low part have at most 26 significant bits in the mantissa. In ExtractScalar a floating-point number is split relative to σ, a fixed power of 2. The higher and the lower part of the splitting may have between 0 and 53 significant bits, depending on σ.

Using Split, the following multiplication routine by G.W. Veltkamp (see [9]) can be formulated.

Algorithm 4 Error-free transformation of the product of two floating-point numbers.

```
function [a, b] = TwoProduct (x, y)
    a = fl(x · y)
    [x_1, x_2] = Split(x)
    [y_1, y_2] = Split(y)
    b = fl(x_2 · y_2 − (((a − x_1 · y_1) − x_2 · y_1) − x_1 · y_2))
end
```

3. Proposed Method

3.1. Format

A number of proposed format is an unevaluated sum of three floating-point numbers in $\mathbb{F}_{50}$ and one floating point number in $\mathbb{F}$. For $a_0, a_1, a_2 \in \mathbb{F}_{50}$ and $a_3 \in \mathbb{F}$, the number (a_0, a_1, a_2, a_3) represents the exact sum $a = a_0 + a_1 + a_2 + a_3$.

Then we require that the quadruple (a_0, a_1, a_2, a_3) to satisfy

$$a_0 \in \mathit{faithful}_{50}(a) \tag{10}$$
$$a_1 \in \mathit{faithful}_{100}(a) - a_0 \tag{11}$$
$$a_2 \in \mathit{faithful}_{150}(a) - a_0 - a_1 \tag{12}$$
$$|a_3| < 2\mathbf{u}_{50}^3 \cdot \mathrm{ufp}(a_0) \tag{13}$$

Note that a_0 is a 50-bit faithful rounding of a.

Lemma 1
Let $a = (a_1, a_2, a_3, a_4)$ be a number of proposed format, $|a_k| < 2\mathbf{u}_{50}^k \mathit{ufp}(a_0)$.

In Hida *et.al.*'s paper, they have proposed "a quad-double number" by an unevaluated sum of four floating-point numbers in $\mathbb{F}$. Proposed format in this paper is similar to Hida *et.al.*'s algorithm, but there are several differences between them. One, the accuracy of proposed format is slightly less accuracy than quad-double arithmetic: the number of mantissa of the proposed format is at least 203 bits, on the other hand, that of quad-double arithmetic is at least 212 bits. Second, the way to store numbers is different. Third, quad-double arithmetic is unique for a real number r, but the proposed format has many way to represent the number.

Most of the algorithms described in this paper produce an expansion that is not of canonical form - often having overlapping bits. Therefore, we present a renormalized algorithm to proposed format satisfying (10) – (13) .

Algorithm 5 A renormalized algorithm

function $[a_0, a_1, a_2, a_3] = \mathrm{Renormalize}(x_0, x_1, x_2, x_3)$
$[s_0, t_0] = \mathrm{TwoSum}(x_0, x_1)$
$[a_0, b_0] = \mathrm{ExtractScalar}\left(2^{53-50} \cdot \mathrm{ufp}(s_0), s_0\right)$
$[s_1, t_1] = \mathrm{TwoSum}(fl(b_0 + t_0), x_2)$
$[a_1, b_1] = \mathrm{ExtractScalar}\left(2^{53-100} \cdot \mathrm{ufp}(a_0), s_1\right)$
$[s_2, t_2] = \mathrm{TwoSum}(fl(b_1 + t_1), x_3)$
$[a_2, b_2] = \mathrm{ExtractScalar}\left(2^{53-150} \cdot \mathrm{ufp}(a_0), s_2\right)$
$a_3 = fl(b_2 + t_2)$
end

We present a following theorem.

Theorem 1
We define $\mathbb{F}(2^m : 2^n)$ by a set of all floating point numbers consisted by bits from 2^n to 2^m ($m > n$). If a sequence $x_0, \cdots, x_3$ satisfying

$$x_0 \in \mathbb{F} \tag{14}$$
$$x_1 \in \mathbb{F}\left(2^{-47}\mathit{ufp}(x_0) : 2^{-99}\mathit{ufp}(x_0)\right) \tag{15}$$
$$x_2 \in \mathbb{F}\left(2^{-97}\mathit{ufp}(x_0) : 2^{-149}\mathit{ufp}(x_0)\right) \tag{16}$$
$$x_3 \in \mathbb{F}\left(2^{-147}\mathit{ufp}(x_0) : 2^{-1022}\right) \cap \mathbb{F} \tag{17}$$

is given. Then the result sequence a_0, a_1, a_2, a_3 of **Algorithm 5** *satisfies the format* (10) – (13)*, and the maximum error of* **Algorithm 5** *is*

$$|x_0 + x_1 + x_2 + x_3 - (a_0 + a_1 + a_2 + a_3)| \leqq 2 \cdot 2^{-203} \cdot \mathit{ufp}(a_0). \tag{18}$$

3.2. Multiplication

Multiplication is basically done in a straightforward way, multiplying term by term and accumulating. Note that unlike addition, there are no possibilities of massive cancellation in multiplication.

First, to achieve the proposed form, we need to modify the multiplication algorithm TwoProduct for m-bit floating point numbers. For a pair of m-bit floating point numbers (x, y) with $x, y \in \mathbb{F}_m$ into a new pair (a, b) with $a, b \in \mathbb{F}_m$ satisfying

$$x \times y = a + b, \qquad |b| \leq \mathbf{u_m}\,|a|\,. \tag{19}$$

Algorithm 6 m-bit TwoProduct. ($27 < m < 53$)

function $[a, b] = \mathrm{TwoProduct}_m(x,\ y)$
$z = fl(x \cdot y)$
$[x_1, x_2] = \mathrm{Split}(x)$
$[y_1, y_2] = \mathrm{Split}(y)$
$w = fl(x_2 \cdot y_2 - (((z - x_1 \cdot y_1) - x_2 \cdot y_1) - x_1 \cdot y_2))$
$[a, v] = \mathrm{ExtractScalar}(2^{53-m} \cdot \mathrm{ufp}(z),\ z)$
$b = fl(w + v)$
end

Theorem 2
For a number $x, y \in \mathbb{F}_m$, the result of **Algorithm 6** *satisfies* (19).

Let $x = (x_0, x_1, x_2, x_3)$ and $y = (y_0, y_1, y_2, y_3)$ be two numbers of the proposed form. Assume (without loss of generality) that a and b are order 1.

$$\begin{aligned} x \times y \quad \approx \quad & x_0y_0 && O(1)\text{ term} \\ & + x_0y_1 + x_1y_0 && O(\mathbf{u}_{50})\text{ terms} \\ & + x_0y_2 + x_1y_1 + x_2y_0 && O(\mathbf{u}_{50}^2)\text{ terms} \\ & + x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 && O(\mathbf{u}_{50}^3)\text{ terms} \\ & (+ x_1y_3 + x_2y_2 + x_3y_1) && O(\mathbf{u}_{50}^4)\text{ terms} \end{aligned}$$

For $i + j < 3$, let $[p_{ij}, q_{ij}] = \mathrm{TwoProduct}(x_i, y_j)$. Then $p_{ij} = O(\mathbf{u}_{50}^{i+j})$ and $q_{ij} = O(\mathbf{u}_{50}^{i+j+1})$. Now there are one term (p_{00}) of order $O(1)$, three (p_{01}, p_{10}, q_{00}) of order $O(\mathbf{u}_{50})$, five (p_{02}, p_{11}, p_{20}, q_{01}, q_{10}) of order $O(\mathbf{u}_{50}^2)$, seven of order $O(\mathbf{u}_{50}^3)$.

In Hida *et.al.*'s paper, there are three different summation boxes. They have called Three-Sum, Six-Three-Sum, Nine-Two-Sum. In proposed multiplication algorithm, in order to reduce the number of floating point operations, we

don't use these summation boxes. Instead of these, we've tried to using normal double precision arithmetic summation. In addition, in order to construct a fast algorithm in this paper, our algorithm don't compute the $O(\mathbf{u}_{50}^4)$ terms; they can affect the first 203 bits only by carries during accumulation. In this case, we can compute the $O(\mathbf{u}_{50}^3)$ terms using normal double precision arithmetic, thereby speeding up multiplication considerably.

Algorithm 7 a multiplication algorithm

function $[a_0, a_1, a_2, a_3] = \text{multiplication}(x_0, x_1, x_2, x_3, y_0, y_1, y_2, y_3)$
$[z_0, b_0] = \text{TwoProduct}_{50}(x_0, y_0)$
$[b_1, c_0] = \text{TwoProduct}_{50}(x_0, y_1)$
$[b_2, c_1] = \text{TwoProduct}_{50}(x_1, y_0)$
$z_1 = fl(b_0 + b_1 + b_2)$
$[c_2, d_0] = \text{TwoProduct}_{50}(x_0, y_2)$
$[c_3, d_1] = \text{TwoProduct}_{50}(x_1, y_1)$
$[c_4, d_2] = \text{TwoProduct}_{50}(x_2, y_0)$
$z_2 = fl(c_0 + c_1 + c_2 + c_3 + c_4)$
$z_3 = fl(d_0 + d_1 + d_2 + x_0 \cdot y_3 + x_1 \cdot y_2 + x_2 \cdot y_1 + x_3 \cdot y_0)$
$[a_0, a_1, a_2, a_3] = \text{Renormalize}(z_0, z_1, z_2, z_3)$
end

Theorem 3
The upper bound of the error of **Algorithm 7** *is* $274 \cdot 2^{-203} \cdot ufp(a_0)$.

4. Numerical results

In this section we present timing of proposed algorithms. We tested in the following environment: Intel Core I7, 1.8GHz, Mac OS X 10.8.2, Memory 32GB. All proposed algorithms were tested in C++. All floating-point operations are done in IEEE standard 754 double precision. We use a compiler `g++ 4.7.2` with `-O2` option as usual. Moreover, to avoid overdoing the compiler optimizations for TwoSum and TwoProduct , an extra compile option `-msse2 -mfpmath=sse` has to be used.

The results for 10 million calls of each algorithms are summarized in Table 1:

Table 1: Measured computing time (sec) and time ratio for 10 million calls of each algorithms

	+/−		×		÷	
Proposed	0.98	(1)	2.00	(1)	8.76	(1)
QD [5]	0.95	(0.9)	3.81	(1.9)	15.3	(1.7)
exflib [2]	2.84	(2.8)	4.75	(2.3)	16.3	(1.8)
MPFR [1]	23.0	(23.)	26.27	(13.)	36.0	(4.1)

5. Concluding remarks

We presented the algorithms and performance of basic operations on the proposed format. Proposed format is an unevaluated sum of four double precision numbers similar to quad-double arithmetic, but it is capable of representing at least 203 bits of significand (50 bit ×3 + 53 bit). Hence, it is slightly less accuracy than a quad-double arithmetic, however, presented algorithms for the format is faster than that for quad-double arithmetic. By numerical experiments it is shown that the algorithms are faster compared to the computation time of the existing algorithm.

Acknowledgements

This paper is a part of the outcome of research performed under a Waseda University Grant for Special Research Projects (Project number: 2012A-508) and this work was supported by JSPS KAKENHI Grant Number 24700015.

References

[1] The GNU MPFR Library: `http://www.mpfr.org/`

[2] exflib - extend precision floating-point arithmetic library: `http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/`

[3] D. H. Bailey, Y. Hida, X. S. Li and B. Thompson. "ARPREC: an arbitrary precision computational package", Lawrence Berkeley National Laboratory. Berkeley. CA94720. 2002.

[4] D. H. Bailey. "A fortran-90 double-double library", Available at `http://www.nersc.gov/~dhbailey/mpdist/mpdist.html`

[5] Y. Hida, X. S. Li and D. H. Bailey. "Quad-Double Arithmetic: Algorithms, Implementation, and Application" October 30, 2000 Report LBL-46996.

[6] S.M. Rump, T. Ogita, and S. Oishi: " Accurate floating-point summation part I: Faithful rounding". SIAM J. Sci. Comput., 31(1):189–224, 2008.

[7] S.M. Rump, T. Ogita, and S. Oishi: "Accurate floating-point summation part II: Sign, K-fold faithful and rounding to nearest". Siam J. Sci. Comput., 31(2):1269–1302, 2008.

[8] D. E. Knuth: The Art of Computer Programming: Seminumerical Algorithms, vol. 2, Addison-Wesley, Reading, Massachusetts, 1969.

[9] T. J. Dekker: A floating-point technique for extending the available precision, Numer. Math., 18 (1971), pp. 224–242.